AFRL-SR-AR-TR-09-0191

REPORT DOCUMENTATION PAGE

1. REPORT DATE *(DD-MM-YYYY)* May 14, 2009	**2. REPORT TYPE** Final Performance Report	**3. DATES COVERED** *(From - To)* March 2006 – November 2008

4. TITLE AND SUBTITLE
An Integrated Specification and Verification Environment for Component-based Architectures of Large-scale Distributed Systems

5a. CONTRACT NUMBER
FA9550-06-1-0223

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
John Hatcliff (PI)
Torben Amtoft (co-PI)
Anindya Banerjee (co-PI)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Kansas State University

Technical Point of Contact	Administrative Point of Contact
John Hatcliff	Paul Lowe
234 Nichols Hall	2 Fairchild Hall
Kansas State University	Kansas State University
785 532 7950	785 532 6804

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR
Dr. David Luginbuhl
david.luginbuhl@afosr.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
There are no restrictions of the distribution of this report.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The objectives of this project are to address challenges of constructing large-scale DoD software intensive systems by constructing an integrated modeling and specification framework that can support software product-line development based on widely-used component middleware frameworks that will likely form the basis of future DoD systems.

This document provides the final performance report on this project.

15. SUBJECT TERMS
Software architecture, verification, component interface specifications, security, secure information flow

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	none	6	John Hatcliff
					19b. TELEPHONE NUMBER *(include area code)* 785 532 7950

An Integrated Specification and Verification Environment for Component-based Architectures of Large-scale Distributed Systems

Award #: FA9550-006-1-0223

Final Report

March 2006 – November 2008

May 26, 2009

John Hatcliff (PI)
Torben Amtoft (co-PI)
Anindya Banerjee (co-PI)

SAnToS Laboratory
Laboratory for Specification, Analysis and Transformation of Software

SAnToS LABORATORY

Department of Computing and Information Sciences, Kansas State University

1 Objectives and Summary of Results

Development challenges for large-scale distributed systems: Large-scale distributed real-time and embedded (DRE) computing systems increasingly form the basis of mission- and safety-critical applications essential to the Department of Defense's (DoD) operational vision. It is tedious, error-prone, and costly to develop, optimize, validate, and deploy these types of DRE systems using conventional software technologies.

Component middleware and modeling as key enablers: A key enabler in recent successes with small- to medium-scale DRE systems (such as avionics mission computing systems) has been *middleware* that provides platform-independent execution semantics and reusable services that coordinate how application components are composed and interoperate. *Component middleware frameworks* like the CORBA Component Model (CCM) and Enterprise Java Beans (EJB) provide additional utility through well-defined interfaces that hide implementations (keeping client code from becoming unnecessarily tangled with low-level implementations in the component) and make code units easier to plug and unplug. In addition, architectural definition languages and meta-modeling environments have enabled software product managers to hide the complexities of dealing with lower-level implementation details by defining structural abstractions of components, interfaces, connectors, and system assemblies that can be visualized, analyzed, and that can drive automatic generation of a variety of forms of infrastructure code.

Limitations and barriers to progress: Despite these successes, a number of barriers are preventing progress of effective verification and validation (V&V) for large-scale DoD systems. *Absence of leverageable semantic content in higher-level architectures and models* prevents significant V&V activities from being carried out at design time – leading to increased development time and costly errors. Despite the utility of *software component technology* for enabling reuse, the *lack of compositional specifications* causes developers to rely on crude notions of component compatibility and composability which prevents rapid component assembly and leads to situations where (a) errors arise due to the composition of components that are not truely semanticly compatible, and (b) composition errors must be detected with costly testing techniques occurring much later in the development process. Even though security issues are increasingly important, *existing security specification and enforcement mechanisms are overly-coarse, inflexible, non-declarative and non-compositional* which makes it very difficult to specify and validate security policies seamlessly in the design of modern component-based distributed systems. Despite the widespread use of software modeling techniques, the *absence of checkable semantic connections between models/architecture and source code* prevents designs and models from serving as faithful abstractions of lower-level implementations that can be leveraged for deep semantic reasoning and V&V.

Our work: Our research on this project addressed the challenges of constructing large-scale DoD software-intensive systems by constructing an integrated modeling and specification framework that supports software product-line development based on widely-used component middleware frameworks that will likely form the basis of future DoD systems. Our approach centered around the following three innovations:

- **A powerful and flexible architecture modeling framework** that builds on our earlier DoD-funded Cadena environment to provide rich type systems that capture important semantic properties directly in component-based architectures,
- **A logic-based compositional specification and verification framework** for secure information flows/dependences in component-based systems, and
- **Code-level static analysis techniques that allow developers to discover and browse crucial information flows in large systems** – leading to more effective engineering of safe and secure systems.

Industrial partnerships and technology transfer: This project involved substantial collaboration with multiple industrial partners with deep ties to the US Air Force and Department of Defense.

- **Lockheed Martin Advanced Technology Laboratory**, *Cherry Hill, New Jersey.* For two years (2006,2007) during the period of performance of this grant, Lockheed Martin funded this project's PIs ($190K total) to apply the basic research and software tools developed under this grant to model-based development and testing of large-scale systems.

- **Rockwell Collins Advanced Technology Center**, *Cedar Rapids, IA*. Throughout the period of performance of this grant, Rockwell Collins funded the PIs ($240K total) to apply the logic and static-analysis-based techniques for reasoning secure information flow to development of network security research sponsored by NSA.
- **Microsoft Research**, *Redmond, Washington*. co-PI Anindya Banerjee spent six months of his 2007-2008 sabbatical at Microsoft Research. While there, he worked with leading researchers in the area of contract-based software verification to help transition some of his work on formal reasoning about object-oriented languages into emerging technologies that are flowing into Microsoft development environments. This work produced several papers jointly authored by Banerjee and Microsoft engineers.
- **IBM, T.J. Watson Research Laboratory**, *White Plains, NY*. Banerjee spent the other six months of his 2007-2008 sabbatical at IBM's Watson Research Laboratory. While there, he worked with leading researchers in the area of security for object-oriented languages to further develop his work on language-based security for object-oriented languages. Banerjee and IBM engineers.

Impact: Our work has produced a number of high-quality publications at top conferences and journals, invitations for keynote talks at international venues, software design and verification tools that have been downloaded 3500+ times during the course of this grant, and industrial technology transitions that are likely to have a lasting impact in technology areas relevant to the Air Force.

2 Summary of Findings

2.1 Indus and Java program dependences

The Indus framework is the only publicly available slicing framework for concurrent Java. Since its initial release in Sept 2004, it has been downloaded over 7500 times by individuals in over 52 countries, with 3000+ downloads since March, 2006. Feedback on the initial and subsequent releases suggest that there is a significant interest in the type of capabilities provided by Indus. Indus has been used in industrial research projects at Fujitsu and Aonix, and its the basis of industry funding provided to KSU by Lockheed and Rockwell Collins.

The AFOSR funding provided on this grant has helped us make Indus available as a resource to the community. Within the past year, several papers have been published by other research groups in top conferences such as ACM Foundations and Software Engineering (FSE) and ETAPS Tools for Construction and Analysis of Systems (TACAS) in which Indus was a primary component of the research. The PI (Hatcliff) has been invited to give the keynote address on the Indus framework at the IEEE SCAM (Source Code Analysis and Manipulation) Workshop in Philadelphia in September 2006. In May 2007, Indus was demoed to security certification authorities at NSA as the basis of a IDE-integrated security analysis framework. Technology developed for Indus is currently being transitioned into development environments for application to security certification at Rockwell Collins.

The technical results and rigorous theory for programming language dependence in our paper "A New Foundation For Control-Dependence and Slicing for Modern Program Structures" ((Below)). Have been used by several other research groups. In particular, Prof. Mark Harman – leader of the Software Engineering Group at King's College, London and director of CREST (Centre for Research on Evolution Search and Testing) – remarked *"I know that this will become one of *the* seminal papers on program dependence and slicing. In fact I am devoting the whole of the next ASTReNet workshop to a discussion of this paper and its consequences."*. Publications 1,2,7 below report on this work.

2.2 Logic-based Specification and Verification of Secure Information Flow

The Multiple Independent Levels of Security (MILS) supported by NSA and the Air Force calls for systems to be developed on top of a "separation kernel" which guarantees isolation and controlled communication between application components deployed in different partitions of the kernel. Our experience with MILS system development at Rockwell Collins confirms that there is a lack of (a) automated tool support for specification and verification of end-to-end MILS policies, (b) mechanisms for producing evidence that policies are enforced.

To address these problems, a central line of work funded by this grant has been to develope a powerful Hoare-logic for reasoning about secure information flow (SIF-logic). The base line for our work (appearing just as this grant was awarded) was a paper by co-PIs Amtoft and Banerjee in the prestigious ACM Principles of Programming Languages

(POPL) conference in 2006. We extended that work with a paper in the 2007 ACM Workshop on Formal Methods in Security Engineering (describing extensions to handle conditional information flow). We worked with Rockwell Collins engineers to design and implement a complete MILS policy language with this Hoare-logic providing the core semantics. Our investigations of real systems identified several key features: the language is compositional (necessary, since MILS is inherently a component-oriented approach), it is developer-friendly (we have been able to integrate a preliminary version directly with SPARK Ada information flow specification and checking used in several large-scale information assurance applications including those at Rockwell Collins), and it is expressive (capturing condition-based information flow and reasoning about heap data that cannot be handled in, e.g. SPARK). We believe that logic-based framework can unify MILS property specification and help automate certification of MILS component integration, e.g., in the Common Criteria MILS Integration Protection Profile, by acting as a formal and checkable "lingua franca" for MILS information flow policies. Publications 3,5,6,13, and 16 below report on advances to foundations for reasoning about secure information flow.

2.3 Formalizing Component-Based Software Architectures

Maintaining integrity, consistency, and enforcing conformance in architectures of large-scale systems requires specification and enforcement of many different forms of structural constraints. While type systems have proved effective for enforcing structural constraints in programs and data structures, most architectural modeling frameworks include only weak notions of typing or rely on first-order logic constraint languages that have steep learning curves and that become unwielding when scaling to large systems.

Publication 8 reports on the Cadena Architecture Language with Meta-modeling CALM – that uses multi-level type systems to specify and enforce a variety of architectural constraints relevant to development of large-scale component-based systems. Cadena is a robust and extensible tool that has been used to specify a number of industrial-strength component models and applied in multiple industrial research projects on model-driven development and software product lines. One of the goals of this research project was to integrate the compositional security contract framework discussed above with Cadena/CALM to achieve a model-based architecture framework with security contracts. One of the Ph.D. students supported on this grant, Edwin Rodriguéz is addressing that problem in his Ph.D. that should be completed in Summer 2009.

2.4 Compositional Interface Checking for Java

Our vision of reasoning about components and their composition includes developing component interface specification languages and verification frameworks for checking that component implementations conform to their interface specifications. Despite its introduction over 30 years ago, symbolic execution has received renewed interest as an analysis framework for checking component interfaces, and for testing and bug finding. One key advantage of symbolic execution over real/concrete execution (e.g., traditional testing) is that one can avoid the burden of constructing numerous concrete input parameter values and instead use symbolic values and constraints to compactly represent sets of possible input values. Significant progresses have been made on using symbolic execution to generate unit test suites, but existing frameworks are still struggling with several issues including poor performance on heap data, lack of coverage goals tied to the space of heap configurations, and lack of support for dealing with open object-oriented systems and generating mock objects.

We have pursued a line of work that seeks to demonstrate how a static analysis feedback and unit test case generation framework, KUnit, built on the Bogor/Kiasan symbolic execution engine addresses the challenges above by: (a) providing an effective unit test case generation for sequential heap-intensive Java programs (whose computation structures are incomplete – open systems), (b) showing how the scope and cost of Kiasan/KUnit's analysis and test case generation can be controlled via notions of heap configuration coverage, and (c) leveraging method contract information to better deal with open object-oriented systems and to support automatic mock object creation.

Papers 9-10 report on this framework as well as symbolic execution importance in the area of software model checking. In a broad experimental study on twenty-two Java data structure modules, we show that KUnit is able to: (a) achieve 100% feasible branch coverage on almost all methods by using only small heap configurations, (b) improve on competing tools for coverage achieved, size of test suites, and time to generate test suites.

3 Personnel Supported

- John Hatcliff (PI), faculty summer salary
- Torben Amtoft (co-PI), faculty summer salary
- Anindya Banerjee (co-PI), faculty summer salary
- Georg Jung, (PhD student), graduate research assistant
- Edwin Rodriguez, (PhD student), graduate research assistant

4 Peer-reviewed Publications

1. Venkatesh Prasad Ranganath and John Hatcliff, "An Overview of the Indus Framework for Analysis and Slicing of Concurrent Java Software (Keynote Talk - Extended Abstract)". Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), September 2006, pp. 3–7, IEEE Computer Society.
2. Venkatesh Prasad Ranganath and John Hatcliff, "Slicing Concurrent Java Programs using Indus and Kaveri", *Journal of Software Tools for Technology Transfer* (Springer), 9 (5) October 2007, pp. 489-504.
3. Anindya Banerjee and Roberto Giacobazzi and Isabella Mastroeni. "What you lose is what you leak: Information leakage in declassification policies." In Proceedings of the Twenty-third Conference on Mathematical Foundations of Programming Semantics (MFPS), April 2007.
4. Anindya Banerjee and David A. Naumann and Stan Rosenberg. "Towards a Logical Account of Declassification (Short Paper)." In Second Workshop on Programming Languages and Analysis for Security (PLAS) 2007 , San Diego, California, June 2007.
5. Marco Pistoia and Anindya Banerjee and David A. Naumann. "Beyond stack inspection: A unified access-control and information-flow security model." In Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P), pp. 149–163, May 2007. IEEE Computer Society Press.
6. Torben Amtoft and Anindya Banerjee. "Verication condition generation for conditional information ow". In 5th ACM Workshop on Formal Methods in Security Engineering (FMSE), 2007.
7. Venkatesh Prasad Ranganath, Torben Amtoft, John Hatcliff, Anindya Banerjee, Matthew B. Dwyer, "A New Foundation For Control-Dependence and Slicing for Modern Program Structures." ACM Transactions of Programming Languages and Systems. Volume 29, Issue 5, August 2007.
8. Georg Jung and John Hatcliff. "A Type-centric Framework for Specifying Heterogeneous, Large-scale, Component-oriented, Architectures", Proceedings of the 6th International Conference on Generative Programming and Component Engineering, Salzburg, Austria. October, 2007. pp. 33–42.
9. "Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems", Xianghua (William) Deng, Robby, John Hatcliff. Proceedings of the 2007 International Conference on Testing: Academic and Industrial Conference: Practice and Research Techniques, IEEE Press, September 2007.
10. "Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs", Xianghua Deng, Robby, and John Hatcliff. Proceedings of the 2007 International Conference on Software Engineering and Formal Methods, September 2007.
11. Matthew B. Dwyer, John Hatcliff, Robby, Corina S. Psreanu, Willem Visser."Formal Software Analysis : Emerging Trends in Software Model Checking", Future of Software Engineering Track. Proceedings of the 2007 International Conference on Software Engineering (ICSE 2007). May, 2007.
12. Anindya Banerjee, David A. Naumann and Stan Rosenberg. "Regional Logic for Local Reasoning about Global Invariants." In Proceedings of the Twenty-second European Conference on Object-oriented Programming (ECOOP) July 2008.

13. Anindya Banerjee, David A. Naumann and Stan Rosenberg. "Expressive Declassification Policies and their Modular Static Enforcement." In Proceedings of the 2008 IEEE Symposium on Security and Privacy, May 2008.

14. Anindya Banerjee, Mike Barnett and David A. Naumann. "Boogie Meets Regions: a Verification Experience Report." In Second IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08). October 2008.

15. Torben Amtoft, "Slicing for modern program structures: a theory for eliminating irrelevant loops", Information Processing Letters, 2008, Vol. 106, pp. 45-51. (DOI: http://dx.doi.org/10.1016/j.ipl.2007.10.002)

16. Torben Amtoft, John Hatcliff, Edwin Rodriguez, Robby, Jonathan Hoag, and David Greve, "Specification and Checking of Software Contracts for Conditional Information Flow", Proceedings of the 2008 International Conference on Formal Methods (FM '08), LNCS 5014, May 2008.

5 Interactions and Technology Transitions

5.1 Collaboration with researchers from Rockwell Collins

Rockwell Collins has established itself as a leader in the construction and certification of MILS-based systems. As stated earlier, a *separation kernel* is the foundation of the MILS architecture. Rockwell's AAMP7 processor provides a micro-code based separation kernel. With funding from NSA, researchers at Rockwell Collins Advanced Technology Center (RC ATC) certified the AAMP7 separation policy – providing the first certified separation kernel. Subsequently, RC ATC engineers verified the separation policy of the Green Hills Integrity 178B RTOS (official award of certification is pending), which is the operating system used in a number of DoD platforms including Lockheed Martin's F-35 Joint Strike Fighter. Based on this ground-breaking work, Rockwell Collins is now building several information assurance products (involving over 200+ developers) on top of the AAMP7 following the MILS architecture.

Rockwell Collins ATC has sought out and funded Kansas State as a research partner based on a belief that Kansas State's expertise in static analysis could be applied to automate and scale-up portions of the verification process. Indeed, early results in a collaborative investigation of the Rockwell Collins verification infrastructure and Kansas State static analysis framework indicate that many elements of the verification framework can be automated leading to: (a) a dramatic reduction in effort, (b) decreased time required to certify, and (c) the ability to handle more complex systems that place a much greater emphasis on concurrency and heap-allocated data.

Given the fact that Rockwell Collins has one of the few certified foundations necessary for supporting MILS and has extensive ongoing MILS development, our team (via its significant collaboration efforts with RC ATC) has an excellent context for studying and advancing MILS technology and its application in large-scale system development. In particular, Rockwell Collins is planning a software product-line approach to produce multiple categories of information assurance devices. The manual aspects of current RC ATC certification will not be able to keep pace with the volume or scale of these systems. Thus, ongoing collaboration between RC ATC and KSU to automate generation of evidence of trust/dependability is crucial.

5.2 Collaboration with researchers at Lockheed Martin

Kansas State University researchers are seeking to dramatically improve MILS development and certification by applying automated static analysis and verification technology whose development is currently supported by Lockheed Martins Software Technology Initiative (STI). STI is an IRAD program administered through Lockheed Martins Advanced Technology Laboratory, and funded by Lockheed Martins Integrated Systems and Government Systems division. The STI effort aims to develop solutions to quickly integrate large-scale, distributed systems of newly developed, legacy and commercial-off-the-shelf (COTS) software. For the past two years, STI has given seed funding to a handful of university research projects such as the work at Kansas State that Lockheed Martin believes are especially promising and relevant to challenges faced by Lockheed Martin in developing large-scale distributed systems.

For the first year under the STI program (Jan 2006-Dec 2006), STI funded applications of the Cadena architecture framework – one of the key technology components of this grant. For the second year (Mar 2007 – Dec 2007), STI is funding Indus static analysis tools that can mine information flows of large-scale systems and display discovered flows in a variety of visualizations that are effective for developers. Very little tool support exists that directly targets the concerns of MILS development, and we are now aiming to transition their techniques into a MILS-aware development environment.

5.3 Collaboration with researchers from IBM

Banerjee (co-PI) is spent six months of his sabbatical at IBM TJ Watson and will be working with IBM researchers (including Marco Pistoia – the author of several widely-read books on Java security). This collaboration has resulted in several publications listed above related to foundations of secure information. As of September 2007, Banerjee is serving the remaining six months of his sabbatical at Microsoft Research in Redmond.

5.4 Collaboration with researchers from Princeton University

This year, we have joined forces with Andrew Appel and Edward Felten's research groups at Princeton who have gained world-wide recognition for their work on proof-carrying code, infrastructure for trusted computing, and formalization of security policies. Their decade-long effort has produced ground-breaking results in foundations of proof-carrying code, paradigms for trusted computing bases, efficient representation of machine-checked proofs as certificates, policy-enforced linking of untrusted components, and certificate-based authentication.

We are now working to build on this previous work to obtain (a) a framework for representing proofs and evidence certificates for our secure information flow logic that can be verified by third-party tools, (b) a *proof-carrying code* paradigm in which the emitted proof information is focused on secure information flow, and (c) a *certifying compiler* supporting MLS/MILS applications. The idea is to obtain ground-breaking compiling technology that (a) generates not only code, but also proofs of correctness for MLS information flow properties and (b) actually has the ability to formally prove full functional correctness of compilation steps (including compilation of implementation languages, design models, and domain-specific languages for MILS policy and system configuration). This technology can radically increase the trustworthiness of MILS by producing pervasive evidence of component and integration correctness.

6 Type-based Formal Modeling of Component Architectures

6.1 Background

Maintaining long-lived, large, distributed, information and computation systems involves a number of challenges. Overall system functionality must be carefully decomposed and arranged into a modular architecture with precisely negotiated interfaces and a clear, hierarchical, organization. Requirements from multiple stake-holders competing in various dimensions such as separate domains of expertise (*e.g.*, hardware interfacing, network, application logic), different levels of abstraction (*e.g.*, supervision, team management, implementation), individual stages of development (*e.g.*, integration of legacy-code, new implementation) *etc.*, have to be systematically reconciled while incrementally adding concerns to the system architecture, a process which continually grows more and more complex throughout the system evolution. Architecture models have to be accurate and robust, *i.e.*, their elements have to faithfully reflect capabilities of the system's execution environment, cater to the abstractions used by various domain experts, and adapt to architecture refinement, globally as well as in detail, while maintaining overall integrity.

The architectural integrity and internal consistency of long-lived, large-scale, projects face many threats, starting with initial design and continuing throughout the system life-cycle. Industrial experience reports indicate a serious need for tools and processes that (a) enable concise, rigorous specification of architectural constraints and (b) provide mechanical checking of conformance to architecture constraints and ensure consistency between various architecture aspects [18, pp. 477-478]. This is even more the case in the context of a *product line* approach, where the degree of cost savings is directly tied to the ability to constrain and impose discipline on architecture elements to increase their potential for re-use over multiple related projects.

At the programming language level, *type systems* have proven to be a very effective paradigm for enforcing constraints on interaction of system units (*e.g.*, class/method types must be compatible with their use), for ensuring that data structures conforming to certain structural invariants (*e.g.*, tree shaped, list shaped), and for characterizing requirements for converting data between different formats. While previous work on architectural definition languages (ADL) and meta-modeling frameworks (frameworks for creating domain-specific modeling languages and environments) has made significant strides toward supporting higher-level architecture development tasks involving specification of architecture units (*e.g.*, components and subsystems), composition of those units, and interactions between units, many existing ADLs use weak type systems and incorporate only limited forms of type checking. Some existing frameworks that have been designed for architecture exchange [25, 21] defer type checking to other tools or provide external constraint languages [83, 57] based on first-order logic that, while powerful, are sometimes difficult for engineers to understand, require verbose definitions to capture simple forms of type checking, become unwieldy and hard to manage as systems scale. Finally, existing ADLs often fail to support several important capabilities needed for large-scale system development including the ability to (a) specify domain or platform specific languages for building open-ended collections of component and interface types, (b) incorporate multiple component models within a single system (as often needed when multiple systems are integrated to form a "system of systems", or for describing multiple levels of abstraction within a system), (c) specify relationships between architectural layers in multi-layered systems, and (d) flexibly combine and extend architectures as system development unfolds.

In this work, we introduce CALM, a type-centric framework for rigorous meta-modeling and architecture definition of component-oriented systems. CALM enables rapid specification and scalable checking of many common forms of architectural constraints that occur in the context of large-scale system development.

The specific contributions of our work are as follows.

- We describe how CALM can specify industrial component models, component middleware platform capabilities, and domain-specific component modeling languages in a rigorous, mechanically leverageable meta-model. CALM also provides operations on meta-models that allow platform descriptions to be flexibly combined and, for example, arranged in inheritance hierarchies to describe refinements of architectural platforms.
- We summarize the foundations of CALM's multi-tiered type system and we explain how (a) the type system enables architects to concisely specify important notions of architecture consistency and (b) how associated type checking enforces compliance of system architecture elements to type-based constraints, domain-specific modeling languages, and platform descriptions captured via typed meta-models.
- We illustrate how the CALM meta-models can be used to address a number of challenges in large-scale architecture development including modeling of heterogeneous systems that include different component models within the same system, using type-based coercions to model capabilities for integrating different platforms, and capturing complex system layering and subsystem abstractions in which elements from one domain are embedded inside of elements from a different one.
- We provide a novel form of typing for *networks of components* that enables a rigorous approach to forming architectural abstractions in which whole systems form the implementation of higher-level architectural elements. The type system can capture completeness or incompleteness of component networks according to their meta-model constraints and supports incremental refactoring of specifications towards complete models.

CALM concepts are implemented in an IBM Eclipse-based framework called CADENA – a robust and extensible environment for modeling and development of component-based systems which is freely available for download [15]. While the generality and expressiveness of CALM has been demonstrated by using it to capture a number of realistic component models (*sec.* 6.7), in this work we illustrate the principles of CALM using a system phrased in terms of a hybrid component model which integrates three architectural styles, including a style for nesC – a component model and associated infrastructure that has been widely used for building wireless sensor networks [27].

The current version of CADENA/CALM has been partially funded and used by Lockheed Martin Advanced Technology Laboratory (ATL) to evaluate the effectiveness of advanced architecture tools as part of their internally funded Software Technology Initiative that seeks to develop innovative technologies for tackling challenges in large-scale system design and integration.

6.2 Principles of CALM

Following previous work on ADLs [56], CALM's modeling primitives are based on the four fundamental categories of entities that define every component-based system: *components* – loci of computation, *interfaces* – loci of interaction, *connectors* – loci of communication, and *configurations* in which instances of components, interfaces, and connectors are allocated and connected together to form what has been termed a *component assembly* or *system scenario*. Existing ADLs tend to organize the definition and use of these elements using two modeling tiers: in the upper tier, developers define component, interface, and connector types, and in the lower tier instances of the declared types are allocated and connected to form component assemblies. Both the upper and lower tiers in existing frameworks (especially those designed for architectural exchange [25, 21]) tend to be very unconstrained since they seek to allow a variety of different component structures to be embedded in them.

	Typing Principle	Benefits
(a)	Instances conform to component types	a component type serves as a template from which a set of component instances can be generated; changes in a component type propagate to all instances
(b)	Typed interfaces on ports	establishes basic notion of protocol/contract on component interaction points
(c)	Type-correct port/interface/role connections	guarantees type compatibility between components
(d)	Types conform to kinds (styles)	guarantees component types and instances conform to vocabulary specified by the style; enables precise specification of component models used in underlying component middleware frameworks and guarantees features of models match capabilities of underlying middleware; enables precise specification of domain-specific component modeling languages
(e)	Architecture style inheritance	enables incremental construction/refinement of architectural styles and platform descriptions
(f)	Incremental typing of component networks	captures if a network of components is completely connected, and if not, summarizes the *potential for connection* corresponding to all unconnected ports; enables a type-checked compositional approach to component assembly and assembly nesting
(g)	Orthogonal nesting of component networks in components and connectors	enables component assemblies to be abstracted either as components or connectors
(h)	Typed-based multi-style nesting constraints	guarantees proper layering of multiple architectural styles within the same system; prohibits developers from violating layering constraints
(i)	Inter-style coercions	captures as formal abstractions the necessary conversions between different platforms and component models in "system of systems" construction

Figure 1: Summary of Cadena type checking capabilities and benefits

In order to more effectively specify and enforce structural constraints, CALM restructures the two-tiered approach to provide three modeling tiers named *style*, *module*, and *scenario* – where each tier constrains and guides activities in the tier below. The style tier is a meta-modeling tier that allows architects to define ADLs constrained to a particular component model or application domain. In particular, CALM styles specify a collection of *type schemas* that give rise to languages of types for building component, interface, and connector types in the module tier below. This approach allows architects to precisely capture the capabilities and type systems available in existing component middleware frameworks like CCM, EJB, *etc.*, and to define *domain-specific* component modeling frameworks. Using the language defined by the style tier, the module and scenario tier provide the two stages of traditional ADLs, *i.e.*, defining the elements of the architecture (module) and instantiating and combining them into assemblies (scenario). Within these two lower levels, the distinction from previous work is that checking inherent in the CADENA implementation guarantees that types declared at the module level conform to an associated style and that instances at the scenario level conform to types. Moreover, the type framework simplifies development by providing palettes and modeling commands tailored to the associated style and module types. While related facilities have been provided in other meta-modeling tools such as GME [50], CADENA provides a variety of additional richer mechanisms for guaranteeing conformance to meta-model and type definitions. Finally, providing a separate style meta-modeling tier enables a collection of novel capabilities that go well beyond simply defining languages of types. Because CALM styles are manipulable artifacts, architects can operate on and define relationships between styles to capture structuring principles relevant for architectural modeling of large-scale systems as illustrated in Section 6.6.

CALM's three-tiered modeling approach is inspired by type theory [65] in which type systems are organized into three levels – *values/instances* conform to *types*, and types conform to *kinds* – and CALM adopts the kinds/types/instances terminology for naming modeling elements in each of its three layers. Figure 1 summarizes the notions of type checking and structural constraint enforcement that are enabled by our approach. We will describe these capabilities in detail in the remainder of the paper, referred to as Capability (a) through (i) (*cap.* (a)–(i)).

6.3 Related Work

Industrial tools: In current industrial practice, initial design is hampered by the inadequacies of existing commercial tools that are almost exclusively UML-based. These tools focus on lower-level class architectures and provide few mechanisms for establishing higher-level architectural subsystem and layering constraints that can guide system architects. Notions of components and interfaces introduced into UML 2.0 are an initial step in addressing these concerns, but tools like Rhapsody and Rational Modeler only provide limited aspects of even the most basic typing capabilities (*cap.* (a)–(c)). Moreover, the generality of UML does not easily allow the architecture vocabulary to be tailored to the concerns of expert designers from different domains working at different layers within the system (*cap.* (d)–(h)) nor are mechanisms provided for formalizing data and interface conversions that mitigate the the inconsistencies and ambiguities that arise as experts from different domains seek simultaneously to express their concerns within the architecture (*cap.* (i)).

Architectural styles: Abowd, Allen, and Garlan [1] proposed the notion of *architectural styles* to capture the environment vocabulary of a software configuration by providing component and connector types, structural constraints, and (optionally) a semantic model [75]. Di Nitto and Rosenblum investigate ADL suitability for modeling component systems, noting the need for support of architectural styles and style refinement [22]. Of ADLs evaluated they found only Acme/Armani [25, 57] satisfactorily supporting style refinement for modeling middleware compliant software through Acme *family* extensions. Nevertheless, an Acme family is simply an enumeration of types (at the level of a CALM module) that form the "palette" from which instances can be drawn to represent a particular style of architecture. There is no higher-level typing mechanism such as *cap.* (d) in Acme to enforce that the types of an Acme family conform to particular constraints on structure or that new types added to the enumeration are aligned with capabilities of a particular execution environment. For example, the Acme user manual notes that "Typically, a family also embodies a set of rules that specify design rules that constrain how designs can be pieced together and declare certain 'well-formedness' rules. However, the Acme type model is actually quite weak, which places a burden on someone defining the family to include either language descriptions about these assumptions, or to specify the constraints in some form that can be interpreted by a tool (*e.g.*, Armani)." [43] Thus, when style constraints are to be enforced, they must be specified and checked by a mechanism external to the type system – with the suggested approach being to use Armani's first order logic (FOL) constraint language. While Acme does support Capability (a), even basic capabilities like (b) and (c) must be specified in first order logic.

CALM goes beyond the notion of families by introducing a separate meta-modeling tier which captures the architectural style in a mechanically leverageable way, thereby defining precisely what can appear within a style and what cannot. While this type-based CALM meta-modeling tier does not provide the same expressive power as first order logic, it is much easier to use, more scalable, and it directly captures most common component system capabilities and structural constraints. We argue that the complexity of first order constraint languages limits the accessibility to developers, making constraints more difficult to specify, maintain, and evolve, while typing on the other hand, being a familiar concept to engineers, seamlessly integrates into development processes and scales easily. We believe that first order constraint languages are necessary, but they should only be applied *after* simpler, more directly integrated, notions of typing are applied. Also, CALM emphasizes the distinction between the meta-modeling tier and the typing and instantiation levels to provide an environment for manipulation, combination, evolution, and cooperation of styles (*cap.* (e)–(i)). These capabilities are not supported in Acme and supporting them in any constraint framework based on FOL seems difficult.

Like Acme, xADL 2.0 has been designed as *architecture exchange language*, but seeks greater robustness and flexibility by using a flexible framework of XML schemas [21]. xADL provides basic notions of component and interface types (*cap.* (b)) but type checking is deferred to other tools. Using xADL XSD schemas [85], all type definitions must reside within element *xArchTypes*, and instances of these types model a run-time system under element *xArchInstance*. However, xADL 2.0 provides no way to strictly separate a platform definition (style) from libraries of design-time types. Using xADL 2.0 it is possible to define a platform vocabulary through a set of types collected under element *xArchTypes*, then provide a schema extension, extending elements representing platform kinds to arrive at a library of types, but this only yields a two-tiered capability similar to that of Acme again, with any conformance checking deferred to other tools. While there is value in a tool engineering approach that enables separate tools to provide constraint enforcement, we wish to pursue a research agenda that enables exploitation of the benefits and synergy that result from directly integrating a variety of forms of typing into the modeling framework itself.

Meta-modeling: The Generic Modeling Environment (GME) is a powerful framework supporting graphic definition of domain-specific modeling languages and the capability to generate domain-specific graphic modeling environments [50, 41]. GME allows users to define a meta-model paradigm using an extended UML class diagram notation with constraints written in OCL. Based on a meta-model paradigm, GME generates a domain-specific modeling environment with entities defined in the paradigm available to graphically construct domain-specific models. GME paradigm definitions like CoSMIC [29] may be written to capture platform component, connector, and interface templates, but some basic notions of type-checking (*cap.* (c)) must be accomplished through OCL constraints. GME offers some support for composing paradigms from elements defined in existing paradigm definitions [51] (*cap.* (e)) but provides no direct support for Capabilities (f)–(i).

Other frameworks: In summary, previous work has provided significant insights and innovations that have inspired our work, and space constraints do not allow a detailed comparison to each of these (for a good survey of ADLs, see [56]). For example, [55] emphasized notions from object-oriented (OO) type systems to describe interesting concepts of component type refinement in the C2 ADL. While C2 supports a particular class of architectures (layered message passing systems) and confines type descriptions to a modeling tier analogous to CALMś module tier, CALM emphasizes use of typing including OO concepts such as inheritance in meta-modeling facilities (at the CALM style tier) that allow one to describe any number of component model styles, style refinement, combinations of styles, nesting relationships between styles.

We seek to complement and add value to previous work by emphasizing a variety of forms of typing to enforce structural constraints. There are other important notions that are orthogonal and can be combined with our approach. Behavioral descriptions [3] and dynamic reconfiguration mechanisms [53] can be added to support specifications of interaction protocols between components. Notations for specifying variability points and variations for software product lines (*e.g.*, as in xADL) can easily be incorporated. In fact, we believe that the strong typing capabilities of CALM are very important for supporting a product-line approach in which variants are plugged into a reference architecture at component and subsystem variability points: CALM's typing at these points serve as a contract on the variation point that potential variants must satisfy to accurately conform to product line architecture.

Previous work on CADENA: The previous version of CADENA [33] was directly tied to CCM and did not include the meta-modeling and architecture structuring capabilities presented in here. A recent article [17] gives a high-level business-oriented summary of the capabilities of the CADENA tool and its use in product-line development. In contrast, the present paper provides technical details for CALM's type system (*cap.* (a)–(e)) and introduces a number of additional capabilities ((f)–(i)).

6.4 Example of a Multi-layer Architecture

Figure 2 presents an example that we use to illustrate various concepts of CALM. A sensor bank consisting of some number of sensors is connected through a local acquisition network to a controller which in turn is linked to a monitor base-station (*fig.* 2(a)). On a lower level of abstraction, the link between the sensor bank and the monitor is established through a radio network link (*fig.* 2(b)). The radio link contains a timer component which is implemented in terms of a nested component assembly (*fig.* 2(c)). Both the Radio Network Link and Timer are built using the nesC sensor network infrastructure and the standard nesC iconography is used in the diagrams of nesC components. Finally, the transmission of the data over a wireless hardware radio link (HWRadioLink) is implemented using a collection of power-controlled, dynamic frequency, phase-key or FM- modulated, hardware components (*fig.* 2(d)).

Although simple, this example includes several characteristics which we believe are intrinsic to architectures of realistic, large-scale systems. We summarize these characteristics and explain how modeling and development of such systems is constrained and guided using the primary typing capabilities of CADENA listed in Figure 1.

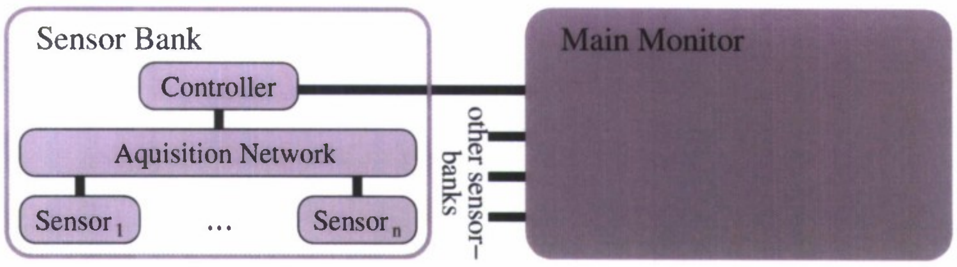
The system includes subsystems built using one or more existing component frameworks (nesC in this case). Therefore, modeling for these subsystems need to be constrained to ensure that capabilities of component framework middleware infrastructure are accurately reflected in architecture specifications (*cap.* (d)). For example, proper capture of nesC capabilities guides developers in building types and component instances that conform to nesC (*cap.* (a) and (c)). Proper capture also ensures accurate mappings between model/specs to code, *e.g.*, it helps ensure that tool infrastructure teams can implement plug-ins that auto-generate code skeletons and deployment "glue code", and import existing nesC libraries into the modeling framework.

The system includes multiple architecture styles within the same system (*e.g.*, the high-level planning style (*fig.* 2(a)), the nesC style (*fig.* 2(b) and 2(c)), and the physical layer style (*fig.* 2(d))) as often required in constructing "systems of systems" that incorporate new, legacy, and off-the-shelf systems built using multiple component frameworks. Architecture specifications need to capture coercions that represent data conversions and marshalling/unmarshalling that often must be implemented to communicate between different component frameworks (*cap.* (i)).

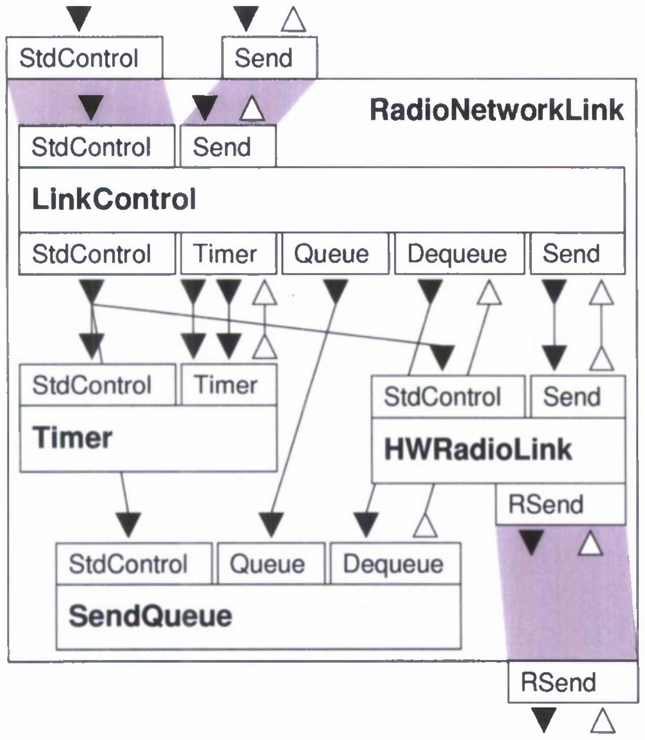
The architecture is layered, and developers must adhere to these layering constraints (*e.g.*, the implementation of each planning layer component must be expressed in nesC and each nesC component associated with the hardware category must be described using the physical layer style) to avoid the architecture degrading over time (*cap.* (h)).

Encapsulation is used as an abstraction mechanism for both components and connectors that may involve a change in architecture styles to represent an abstraction boundary. For example, the HWClock timeout-generator is encapsulated inside of the Timer component of the RadioNetworkLink, and the RadioNetworkLink is encapsulated (with a style change) inside of the Controller/Monitor link of the planning assembly (*cap.* (g)).

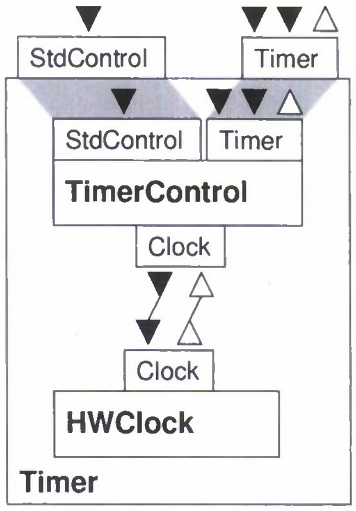
While nesC components initially all have the same shape, in the example it is reasonable to distinguish them



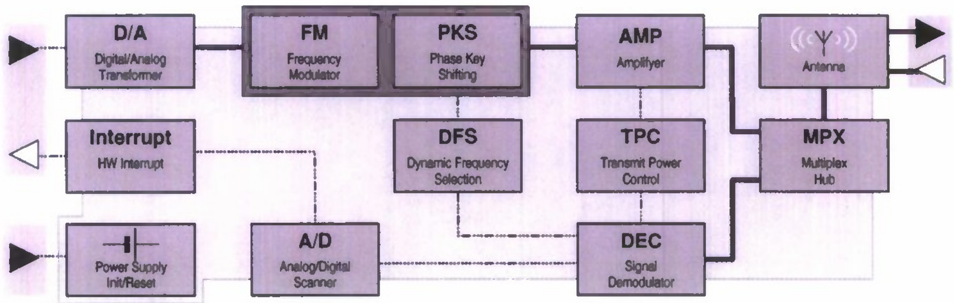
(a) Abstract-planning layer assembly



(b) Data-link layer assembly



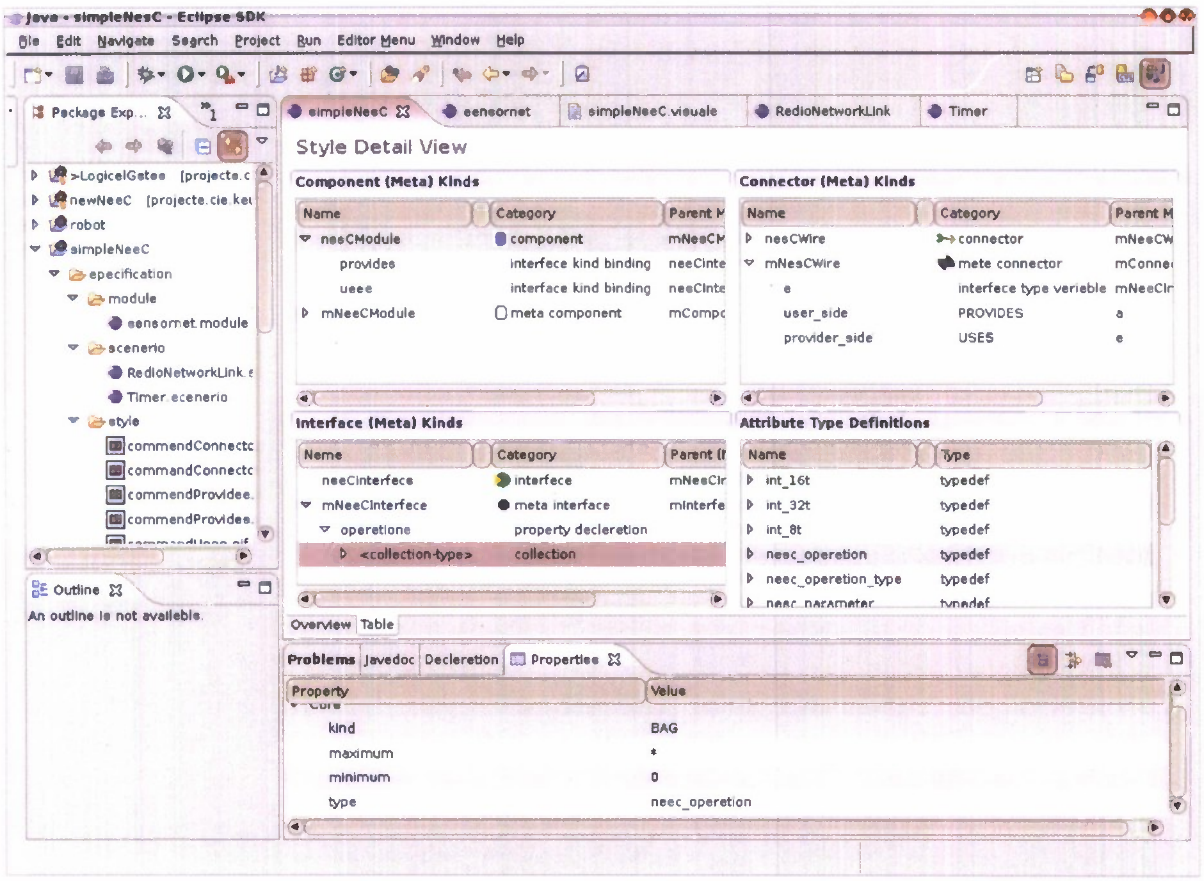
(c) Hardware clock wrapper



(d) Physical layer assembly

Figure 2: Multi-style architecture

according to their possible contents. For example the LinkControl or the SendQueue components are software implementations, while the Timer represents a sub-scenario, and the HWRadioLink abstracts an assembly on the physical network layer. We use *refinement* to enhance the existing nesC style to reflect these differences (*cap.* (e)).



Style Detail View

Component (Meta) Kinds

Name	Category	Parent M
nesCModule	component	mNesCM
provides	interface kind binding	nesCInte
uses	interface kind binding	nesCInte
mNesCModule	meta component	mCompo

Connector (Meta) Kinds

Name	Category	Parent M
nesCWire	connector	mNesCW
mNesCWire	meta connector	mConne
s	interface type variable	mNesCIn
user_side	PROVIDES	s
provider_side	USES	s

Interface (Meta) Kinds

Name	Category	Parent (I
nesCInterface	interface	mNesCIn
mNesCInterface	meta interface	minterfa
operations	property declaration	
<collection-type>	collection	

Attribute Type Definitions

Name	Type
int_16t	typedef
int_32t	typedef
int_8t	typedef
nesc_operation	typedef
nesc_operation_type	typedef
nesc_parameter	typedef

Property	Value
Core	
kind	BAG
maximum	*
minimum	0
type	nesc_operation

Figure 3: The CADENA style editor

6.5 Typed Modeling in CALM/CADENA

We now illustrate how CALM's three tiers are employed to define a modeling language for the nesC component framework used in the example of Section 6.4 and how that modeling language is subsequently applied to define type and instances of nesC components.

The style tier. Figure 4 shows a possible CALM style specifying the nesC component model in a syntax specifically designed for easy presentation in this paper (the CADENA tool-suite relies completely on graphic and form-based input instead, *e.g., fig.* 3). To capture the types that are to be available to developers programming in nesC, Lines 2–19 define a few of the nesC platform-types (since for example purposes only, the list is not comprehensive). Lines 21–38 define *kinds* that represent languages (schemas) of types that can be used to build the nesC interfaces, component, and connectors model entities. An architect first uses CALM *meta-kinds* to construct the basic building blocks for modeling entities. Meta-kinds can be extended through inheritance to facilitate reuse. Once the construction for a particular class of entities is finished, the architect declares an associated kind from a meta-kind, which exposes the type language for use by developers at the module level.

This (simplified) nesC style features one kind of interface, which in turn consists of an arbitrary number of *events* (*i.e.*, asynchronous messages from the provider of an interface to the user), and *commands* (*i.e.*, messages from the user of an interface to the provider).[1] Lines 21–22 define the interface meta-kind `mNesCInterface` containing a list of nesC-operations. Line 22 illustrates CALM's *attribute*. In general, attributes can be associated with each of the CALM architectural entities to capture both the platform types of the targeted middleware or execution environment as well as meta-data types such as organizational data, deployment information, or physical units (*e.g.*, SI-units) not

[1] The original, non-simplified, nesC style has two more kinds of interfaces besides the "bundle": bare command- and bare event-interfaces. They can be omitted without loss of expressiveness.

```
1   style nesC {
      typedef result_t        = ENUM { success, fail };
3     typedef uint8_t         = INT[0..255];
      typedef uint16_t        = INT[0..65536];
5     typedef uint32_t        = INT[0..4294967296];
      typedef int8_t          = INT[-128..127];
7     typedef int16_t         = INT[-132768..32767];
      typedef int32_t         = INT[-2147483648..2147483647];
9     typedef nesc_type       = UNION { result_t, uint8_t,
        uint16_t, uint32_t, int8_t, int16_t, int32_t }
11    typedef nesc_operation_type = ENUM { event, command };
      typedef nesc_parameter = struct {
13      name : STRING, type : nesc_type };
      typedef nesc_operation = struct {
15      async          : BOOLEAN,
        name           : STRING,
17      operation_type : nesc_operation_type,
        parameters     : nesc_parameter list,
19      return_type    : nesc_type };

21    metainterface mNesCInterface {
        attribute operations : MODULE nesc_operation list };
23    interfacekind nesCInterface : mNesCInterface {};

25    metacomponent mNesCModule {
        provides [0..*] provides : mNesCInterface [0..*];
27      uses     [0..*] uses     : mNesCInterface [0..*] };
      componentkind nesCModule : mNesCModule {
29      provides -> nesCInterface;
        uses -> nesCInterface };
31
      metaconnector mNesCWire {
33      uses     [1] provider_side : mNesCInterface [1];
        provides [1] user_side     : mNesCInterface [1] };
35    connectorkind nesCWire : mNesCWire {
        typevar a : nesCInterface;
37      provider_side -> a;
        user_side -> a }
39  }
```

Figure 4: nesC-ADL style

covered by the platform types. In this case, the attribute `operations` captures the ability to define a list of operation signatures built using the nesC platform types declared earlier. CALM attributes can be labeled with *binding times* indicating that the attribute should be bound to a value either at the style level, module level, or the scenario level. In this case, the attribute `operations` is defined to be module-level (*l.* 22), *i.e.*, it describes a property of a *type* within this kind, in contrast to style-level attributes which define properties of the whole kind, or scenario-level attributes which define specifics of an instance. Finally, line 23 defines an interface kind named `nesCInterface`. This kind allows developers at the module level to build interface types that must conform to the structure of meta-interface specification `mNesCInterface`.

NesC provides one kind of component, called *module* (not to be confused with the CALM module tier). A nesC module can provide or use an arbitrary number of interfaces as its ports. In CALM a language for building component types with certain categories of ports is modeled through defining *port-options* in the component (meta) kind (*l.* 26–27). A port-option starts with a parity, either **`provides`** or **`uses`**, to indicate whether or not the interface represents a service that the component provides or a service that the component needs to connect to (*i.e.*, a context dependency). Then, an integer-interval, the *multiplicity*, constrains how many ports within the respective option any component of that kind must/can have. The first port-option of the `mNesCModule` (*l.* 26) defines a minimum of zero ports and no maximum (`[0..*]`). The third position in the port-option is the module-level keyword, which – in accord with the notion of creating a language – is used by developers at the module tier to indicate the particular category of ports being declared. The standard nesC keywords defined in the port-options in lines 26 and 27 (`provides` and `uses`) coincide with the CALM parities, but are conceptually unrelated. Position four specifies the meta-kind from which interfaces associated with this port can be drawn, in this case `mNesCInterface`. Finally, the port-option defines the *multiplexity* that constrains the number of connections that can be made to ports within this option, *i.e.*, the range of minimum and maximum fan-out. When exporting the `nesCModule` component kind from the `mNesCModule` meta-kind (*l.* 28–30), the architect must specify a particular interface kind for each interface meta-kind declared in the port options of the

```
   module sensornet of nesC {
2    nesc_operation init = struct { async = false,
       name = "init", operation_type = command,
4      parameters = [], return_type = result_t };
     nesc_operation send = struct { async = false,
6      name = "send", operation_type = command,
       parameters = [struct { name = "payload",
8        type = uint32_t }], return_type = result_t };
     nesc_operation sendDone = struct { async = false,
   ...
40   nesCInterface StdControl {operations = [init] };
     nesCInterface Send { operations = [send, sendDone] };
42   nesCInterface RSend extends Send { };
     nesCInterface Timer { operations = [start, stop, fire] };
   ...
48   nesCModule LinkControl {
       provides init   : StdControl;
50     provides input  : Send;
       uses reset    : StdControl;
52     uses clock    : Timer;
       uses queue    : Queue;
54     uses dequeue  : Dequeue;
       uses output   : Send }
56   nesCModule Timer {
       provides init  : StdControl;
58     provides timer : Timer }
   ...
     nesCModule TimerControl extends Timer {
70     uses clock : Clock }
   }
```

Figure 5: nesC-types module

associated component meta-kind. In this case, the `nesCInterface` kind is specified for both port options.

The one service featured in nesC is a one-to-one communication service called *wire* which connects interfaces of identical type. Analogous to the port options on component (meta) kinds, connector (meta) kinds have *role options* which specify the ability to define connection points associated with particular kinds of interfaces. Lines 33-34 declare role options for the `mNesCWire` connector meta-kind. In contrast to the port options seen earlier, the multiplicity and multiplexity of both role options are set to `[1]`, shorthand for the interval `[1..1]`, meaning that connectors conforming to this meta-kind can only have one connection point for each of its two roles (*i.e.*, the connector is binary) and that the fan-out value for each connection point is constrained to be one. CALM allows different compatibility requirements to be stated for component interfaces that communicate through a connector. In addition, type variables can be introduced to achieve a notion of polymorphism. For example, to export the `nesCWire` from the meta-kind `mNesCWire`, a type variable `a` for types within the kind `nesCInterface` is declared (*l.* 36). Type variables such as `a` enable CALM to express constraints about relations between types such as equality (=), and sub- or super-typing (>=, <=). In this simple case, in which the wire can only connect interfaces of what would be equal type in the CALM model, the variable `a` is used twice, denoting equality of the types. Therefore, the role `provider_side` and the role `user_side` can both associate with the same type of `nesCInterface`.

The module tier. Having defined type schemas (via CALM (meta) kinds) for nesC types, developers work at the module tier to build up libraries of types, and CADENA tool support guarantees that these types conform to schemas declared at the style tier.

Figure 5 shows excerpts from a CALM module, declaring types in the ***nesC*** style from Figure 4. A CALM module declares types for interface kinds (*e.g.*, *l.* 40–43) and for component kinds (*e.g.*, *l.* 48–70). Connector types are not declared on the module tier, the style-level typing constraints enable connector types to be created on-the-fly when instantiated on the scenario tier. Again, this strategy emphasizes the interpretation of connectors as service entities and as part of the infrastructure.

Note for example the ***nesCModule***-type `LinkControl` (*l.* 48–55). The type is declared with the kind name ***nesCModule*** from the style, ports on the type are declared with the names of the port-options they comply to, *i.e.*, ***provides*** and ***uses***, featuring ***nesCInterface***-types previously declared (*l.* 40–46). The module-level attribute ***operations*** of the ***nesCInterface***-kind is valuated to declare ***nesCInterface***-types (*l.* 2–37, 40–46).

```
scenario timer_assembly includes sensornet {
  TimerControl control { };
  Clock        hwClock { };

  nesCWire { user_side = control.clock;
             provider_side = hwClock.clock }
}
```

Figure 6: Timer assembly in CALM

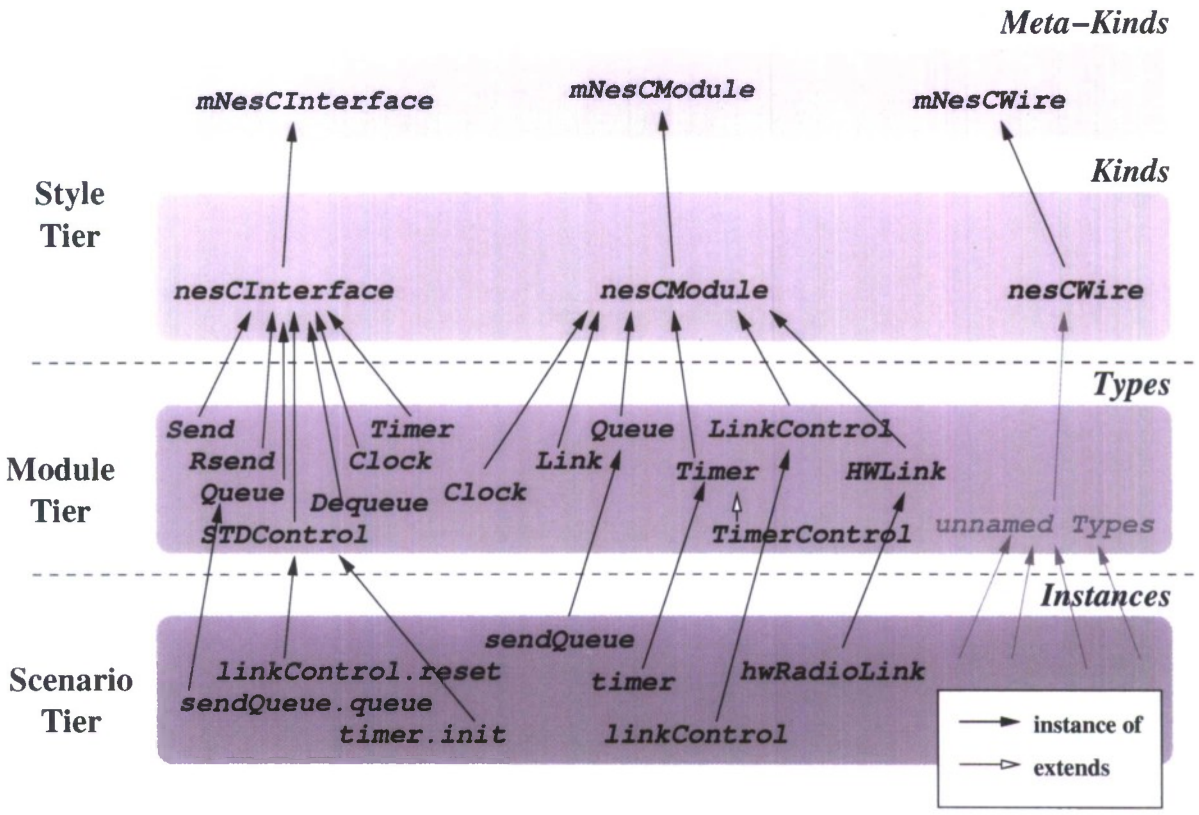


Figure 7: nesC kind-type-value entity-relations in CALM

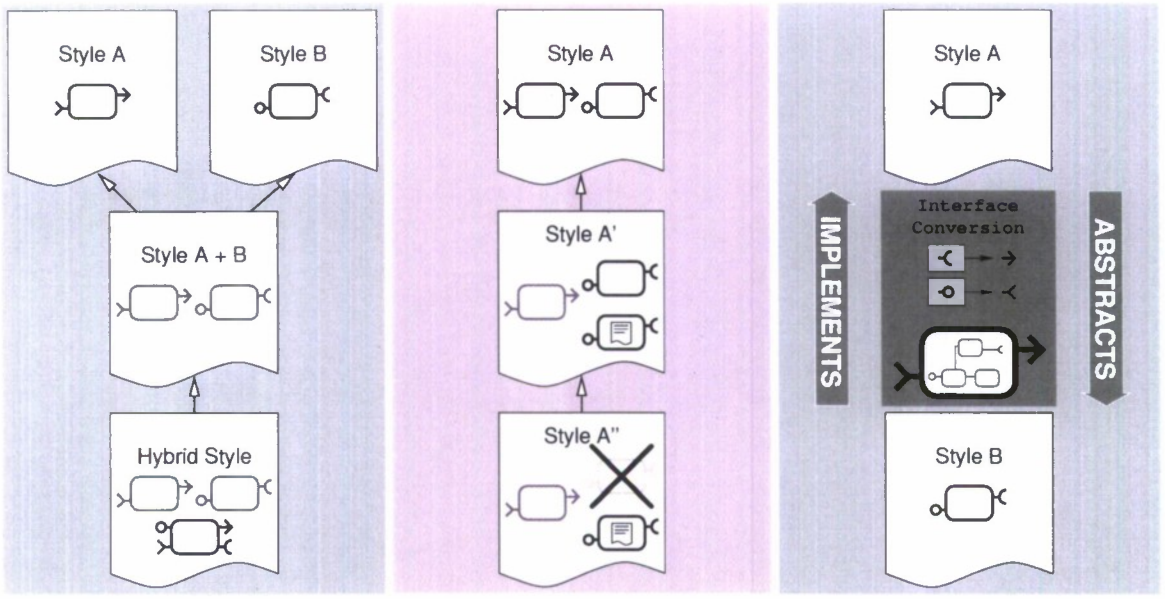
Declaring the types of an architecture means declaring the building blocks or the functional units. While nesC itself does not distinguish clearly between type and instance, CALM models always contain the type layer and each type can be instantiated multiple times.

The scenario tier. Figure 6 shows the assembly of the nesC hardware clock wrapper outlined in Figure 2(c) (Section 6.4). In Line 2 the ***nesCModule***-type ***TimerControl*** is instantiated to obtain the value `control`. Naturally, the provided interfaces of that type, *i.e.*, ***StdControl*** and ***Timer*** are instantiated with the component. A ***nesCWire*** with unnamed type is instantiated in lines 5–6 and connects the two ***nesCModule*** instances with the appropriate ports.

As a summary, Figure 7 visualizes the genesis of model elements through the three tiers of CALM, starting with the meta-kinds and kinds on the style tier, through the types on the module tier down to concrete values on the scenario tier.

6.6 Style Manipulation and Combination

With the architectural style being a separate, manipulable, part of the modeling framework, CALM allows various operations on styles which go beyond architectural exchange and directly address problems of architecture refinement



(a) Generating a Hybrid Style (b) Refining a Style (c) Implementing/Abstracting a Style

Figure 8: Style interrelations

and combination of domain-specific layers in multiple dimensions. Figure 8 gives a conceptual visualization of three different ways to modify and interrelate styles, related to Capability (e) and (f) (*fig.* 8(a), 8(b)) and Capability (g) through (i) (*fig.* 8(c)). This section overviews how emphasizing type-based meta-modeling enables architects to easily combine and refine styles using multiple inheritance. Working at the style tier, architects also (a) define *relationships between styles*, *e.g.*, defining precisely where coercions are available to connect interfaces from different styles and (b) specify *layering/abstraction constraints*. Larger examples and associated formalization can be found in an accompanying technical report [40].

6.6.1 Building hybrid styles

Figure 8(a) illustrates how *hybrid styles* (*e.g.*, representing different platforms cooperating on the same abstraction level) can be formed through CALM style manipulations: a union of two styles is formed (via multiple inheritance), and then "bridging elements" (*i.e.*, component or connector kinds whose ports/roles associate interface kinds drawn from both styles) are introduced. This capability enables system architects to formally capture the practice of integrating components from different sources with related yet dissimilar context requirements (*e.g.*, Bonobo and CORBA components within the Linux Desktop), and allows for a smooth transition of existing scenarios into hybrid environments.

6.6.2 Refining styles

CALM style manipulations allow architects to create modeling environments which control the evolution or refinement of an architecture from a general purpose platform-independent description (as might be used in earlier stages of development) to a style which contains more specifics about the underlying platform. For example, Figure 8(b) illustrates a situation where the architect specifies a style A in which developers initially work. The style is constrained to be platform independent by the absence of any kinds that describe the capabilities of the specific platform. This prevents rogue developers from "running ahead" of the development plan by adding additional details that might threaten the genericity or portability of the system description. Once further details of a target platform are identified, the architect (a) creates a sub-style A' which adds new, more specific kinds that capture *e.g.*, particular communication services available on the target platform as new connector kinds, and (b) directs developers to change generic capabilities of style A to specific implementation options exposed in style A'. Developers carry out these tasks in style

```
style nesC_refined extends nesC {
  elide nesCModule;

  componentkind nesCSoftModule : mNesCModule {
    provides -> nesCInterface;
    uses -> nesCInterface };
  componentkind nesCHWModule : mNesCModule {
    provides -> nesCInterface;
    uses -> nesCInterface };
  componentkind nesCNWModule : mNesCModule {
    provides -> nesCInterface;
    uses -> nesCInterface };
}
```

Figure 9: nesC-ADL style (refined)

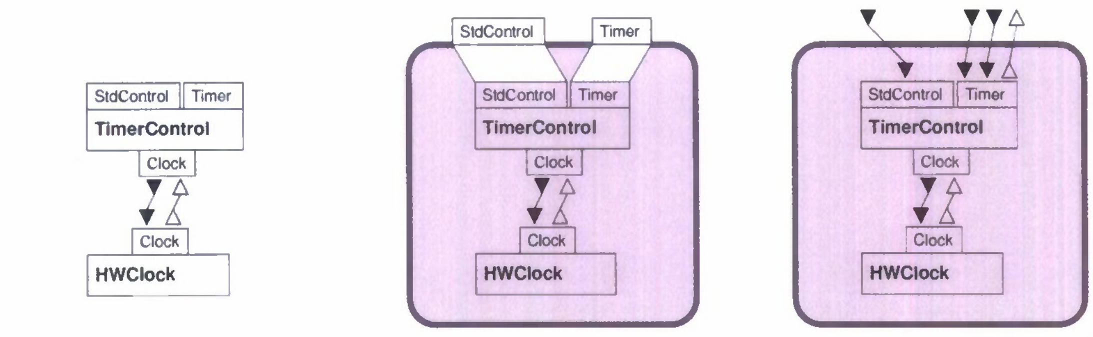
A' which contains modules and scenarios which are not yet completely migrated (the style supports both the original generic elements as well as newly added platform specific elements). As the migration phase nears completion, it can often be quite difficult to tell via manual inspection if all generic modeling elements from style A have been replaced by the more specific elements added to obtain style A' (this is especially true in large-scale development). However, using CALMś automatic type checking, confirmation of a completed migration is obtained simply and efficiently – the completeness of the migration can be validated by type checking the migrated modules against a style A" which is formed from A' by using CALM's *kind elision* operator to remove the original platform independent kinds from A. This same process can be repeated multiple times, forming a succession of validated development "check points" moving from platform independent to increasingly platform specific architectures in a controlled sequence of style refinements.

Figure 9 refines the original nesC style by introducing three new kinds, and at the same time eliding the ***nesCModule*** kind. This new style will be used to connect nesC to other styles by distinguishing the components with respect to their possible implementation contents. The new component kinds ***nesCSoftModule*** (for software-implemented components), ***nesCHWModule*** (for hardware wrappers), and ***nesCNWModule*** (for network-infrastructure wrappers) are not distinguished by the nesC definition, yet the architect can further tailor a component model like nesC to a particular development context, for example to establish layering and nesting constraints such that only modules and scenarios from the hardware style can be nested in components that are built from the ***nesCHWModule*** kind (*sec.* 6.6.3). In [17] the possibilities for model abstraction, specialization, migration, and hybrid construction given through style manipulation are discussed in more detail.

6.6.3 Implementation-abstraction relations on styles

Typing assemblies While various notions of typing for components, interfaces, and connectors at modeling layers analogous to CALM's module and scenario tiers have been considered in earlier works, we are not aware existing approaches for typing assemblies of allocated components as units where the assembly type serves as a summary of how the assembly can interact with its context. CALM includes a notion of typed assembly that captures the *overall connection potential* in terms of may/must modalities (*cap.* (f)). In CALM, a port/role is called *open*, if its multiplexity allows further connections (may connect), otherwise it is called *closed*. A port/role is called *complete*, if its multiplexity does not require any further connection, otherwise it is called *incomplete* (must connect). Intuitively, *completeness* requirements of ports/roles constrain the minimum of further necessary connections in a scenario, while *openness* indicates the maximum of further possible connections. CALM calls the set of possible types which fall into these minimum/maximum constraints the *type-spectrum* of the scenario. In other words, type checking of the assembly not only tells the developer about the compatibility of component/connector connections, it also indicates whether additional connections are possible/required. Those pending connections can be presented in a tool task list along with their corresponding interface types, so that developers can easily identify remaining development steps. For each such type, CADENA presents a summary of type-correct connection opportunities (automatically filtering out incompatible types), and this serves to rapidly focus the developer's attention on appropriate connections.

As an example, consider the scenario `timer_assembly` in Figure 6, Section 6.5. It features two component instances `control` of type ***TimerControl***, and `hwClock` of type ***Clock***. Associated with these components, the scenario contains three provided interfaces, `control.init` of type ***StdControl***, `control.timer` of type ***Timer***, and `hwClock.clock` of type ***Clock***, and one used interface, `control.clock` of type ***Clock***, *i.e.*, the assembly has



(a) the timer assembly (b) as component (c) as connector (service)

Figure 10: Wrapping the timer-assembly

four *ports*. Also, it contains one unnamed instance of a ***nesCWire***-kind connector, through which it features two *roles* (*fig.* 10(a), note that nesC displays single connectors sometimes as "bundles" of lines).

By their multiplexity, all roles in the scenario are closed and complete, because their minimum and maximum fan-out of one connection is used. The ports on the other hand are complete but open (*i.e.*, their multiplexity invariantly is `[0..*]`, they can connect with an arbitrary number of further connectors). Thus, as a maximum, this assembly can be typed as a component with all four aforementioned ports exposed, while as a minimum, no ports or roles have to be exposed. If within this spectrum, the two ports `control.init` and `control.timer` (*fig.* 10(b)) are chosen to be exposed, the resulting type of the assembly is equal to the declared ***nesCModule*** type ***Timer*** (*fig.* 5, *sec.* 6.5).

Alternatively, the type spectrum can be changed by adding two ***nesCWire***-kind connectors attached to the previously exposed ports `control.init` and `control.timer` with the respective ***user_side*** role left open (*fig.* 10(c)). As a result, the two unconnected roles become minimum members of the type spectrum, the assembly can now be typed as a connector by exposing only the two open roles, and hiding the complete ports. Unary connectors such as this one have turned out to be an elegant means to abstract services provided by library functions in nesC such as the timer.

In summary, this notion of typing provides a sound approach to specifying when an assembly can be abstracted as a component or a connector (*cap.* (g)). If all the roles in an assembly are complete (*i.e.*, all connection responsibilities of connectors are fulfilled), then it can be typed as a component with an interface corresponding to the open ports of the assembly. If the ports are complete, the assembly can be typed as a connector, featuring only the open roles. Further, not all ports of a component-typed assembly (or roles of a connector typed assembly) have to be visible in the abstraction, only incomplete ones have to be mentioned.

Abstracting an assembly into a component or connector is accomplished using CALM's *wrapping* facility. For example, the wrapping into a component as described above (*fig.* 10(b)) is accomplished through

```
implementation HWTimer :
  wrap timer_assembly into sensornet.Timer {
    expose init  = control.init;
    expose timer = control.timer };
```

CADENA records such wraps in *implementation tables*, so that when instantiating a component the architect can choose a fitting implementation, which might include simulation stubs if a component is to be placed in a testing environment. Figure 11 shows a screenshot of CADENA with the Timer-assembly associated with the Timer-component in RadioNetworkLink. While other frameworks also include the notion of nesting (almost always limited to just nesting in components), the novelties here are that (a) built-in typing guarantees the well-formedness of the nesting, and that (b) our distinction of open/closed roles and ports enables the principle to be applied orthogonally to nesting in components and connectors. In addition, our assembly types provide a foundation for typing *component architectural patterns* in which assemblies are parameterized via assembly-typed placeholders/variables into which other assemblies of compatible type can be substituted.

Style coercions and nested styles A key feature of CALM is its ability to specify when different styles can be used in the same system model at different levels of abstraction to allow capturing such relations of Figure 8(c) (*cap.* (h)–(i)). Figure 12 shows two revisions of a conceptual style as used for the highest level assembly of this paper's example

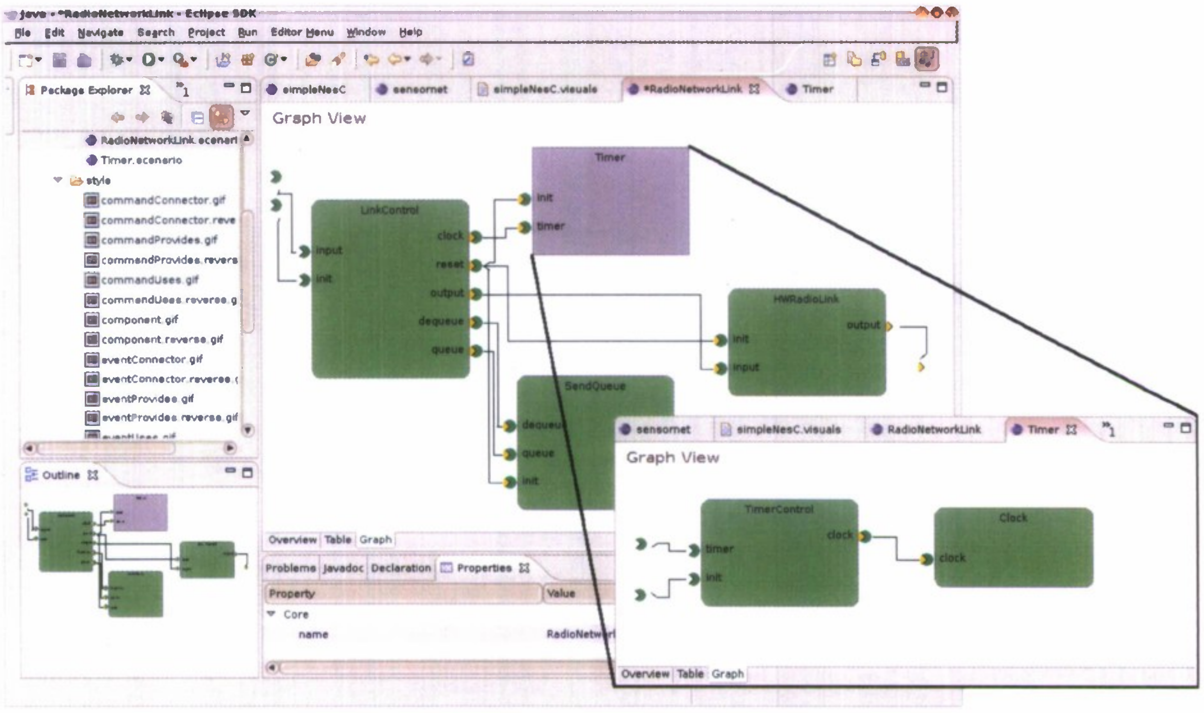


Figure 11: The Timer assembly modeled as component of Radio-Link in CADENA

(a) Global style

```
style global_a {
  metainterface mPlug { };
  interfacekind Plug : mPlug { };

  metacomponent mBox {
    attribute name : SCENARIO STRING;
    provides [0..*] in  : mPlug [0..*];
    uses     [0..*] out : mPlug [0..*] };
  componentkind Box : mBox {
    in -> Plug;
    out -> Plug };

  metaconnector mLink {
    uses     [1] source : mPlug [1];
    provides [1] sink   : mPlug [1] };
  connectorkind Link : mLink {
    typevar a : Plug;
    source -> a;
    sink -> a }
}
```

(b) Global style refined

```
style global_b extends global_a {
  elide Link;
  metaconnector LocalLink : mLink {
    typevar a : Plug;
    source -> a;
    sink   -> a }
  metaconnector RemoteLink : mLink {
    typevar a : Plug;
    source -> a;
    sink   -> a }
}
```

Figure 12: Global style (base & refined)

(*fig.* 2(a), *sec.* 6.4). As stated, many elements of this architecture are implemented in nesC at deeper levels of nesting, *i.e.*, the `global` style *abstracts* `nesC`, and in turn `nesC` *implements* `global`. Specifically, the connector representing the communication link between the Controller and the Main Monitor is realized by an non-trivial nesC assembly (*fig.* 13) in which an instance of the RadioNetworkLink (*fig.* 2(b)) is used at each end of the connector.

CALM enables the architect to capture implementation/abstraction relationships such as the one described here with specifications on each modeling tier: First, the style level defines a *coercion* between the ***global_b*** style and nesC:

```
coercion nesC_to_global :
  from nesC build global_b.Plug {
    provides [1..*] available : nesCInterface
    uses     [1..*] required  : nesCInterface };
```

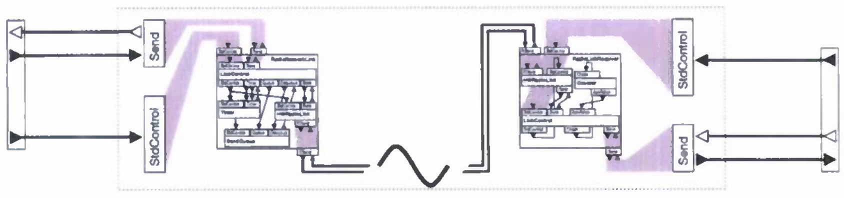


Figure 13: Network-assembly as connector

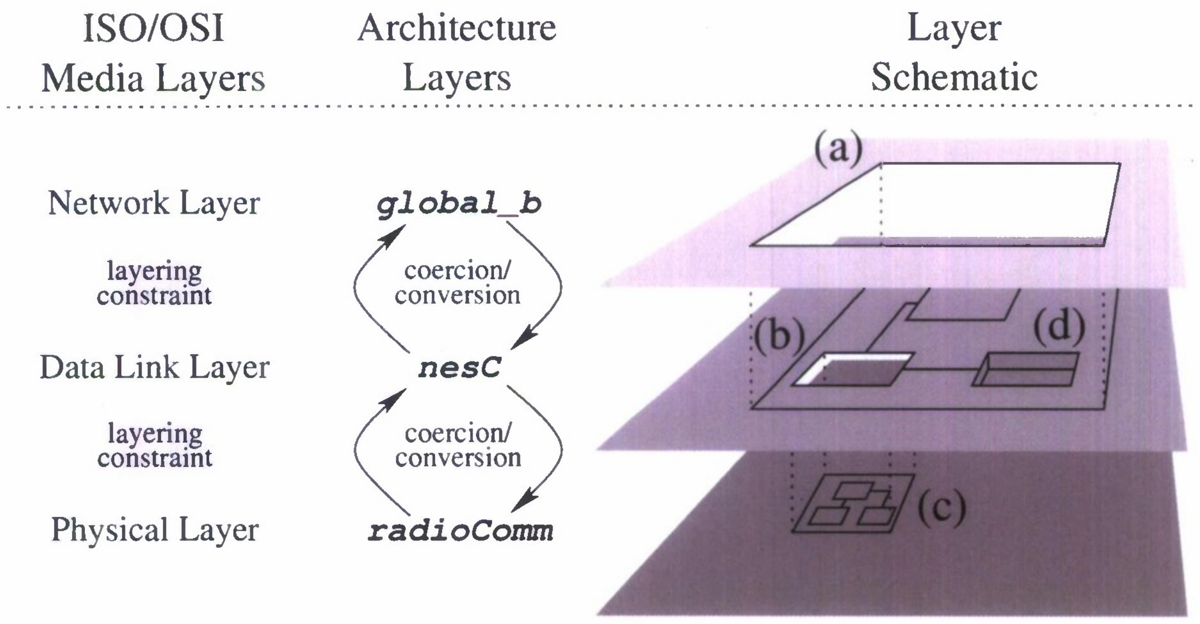


Figure 14: Constraining inter-style nesting

A coercion specifies that interfaces from an abstract overview-style (***global_b***) can be expressed in terms of a set of interfaces from the implementing style, in this case an arbitrary number of elements from ***nesCInterface*** are specified as valid implementation of an element of the ***Plug*** interface kind. A coercion always describes provided interfaces, *i.e.*, a provided interface of the kind ***Plug*** can internally provide interfaces of kind ***nesCInterface*** declared with the `available` keyword introduced in line 3, and use ***nesCInterface***-kind interfaces introduced with `required`. For a ***Plug*** port or role with parity **uses** the internal parities are reversed.

On the type level (module tier), *type conversions* can be declared with the vocabulary defined by the coercion:

```
nesC_to_global sensornet_to_bank :
  abstract sensornet into sensorbank.Send {
    available send : Send,
    available control : StdControl };
```

This conversion packs one ***sensornet.Send*** interface type and one ***sensornet.StdControl*** interface type into the interface type ***sensorbank.Send***. This enables the architect to wrap the network assembly into a connector of the global style analogous to the single-style wraps explained in Section 6.6.3 (*cap.* (g) and (i)).

Multi-dimensional layering Figure 14 interprets the implementation/abstraction relation between CALM styles as positive layering constraints, in the sense that a style A can access functionality of a style B iff A's architectural elements can provide a shell for assemblies of B through wrapping, which is enabled through respective coercions/-conversions of interface kinds/types. In this specific case, architectural elements of the ***global_b*** style (*fig.* 14(a)) are implemented in the ***nesC*** style which represents the layer below the global assembly. The ***nesC*** style in turn has elements implemented by the **`radioComm`** style (*fig.* 14(b) and (c)), and other elements implemented in a nested assembly in **`nesC`** (*fig.* 14(d)).

These conceptual layers, given through specific styles, correspond to the suggested lower (media-) layers of the ISO/OSI stack [38]. The absence of coercions between the styles of the network layer and the physical layer prevents direct access in circumvention of the data-link layer. This use of component encapsulation provides a correct-by-construction approach to layered architectures, as well as the ability to intorduce layers in multiple directions (*i.e.*, specific to individual components).

6.7 Evaluation

We have evaluated the generality of our CALM typed modeling concepts by using the CALM style mechanism to define CADENA modeling environments for multiple "industrial strength" component models including Enterprise Java Beans (EJB) [54], the CORBA Component Model (CCM) [62], Boeing's PRiSM component model, and nesC [26, 27] – demonstrating the ability to handle component frameworks ranging from enterprise-level to real-time (avionics) and embedded systems. Using CADENA's plug-in facility to enhance the basic functionality of the resulting CADENA CCM modeling framework, we have created an end-to-end model-based development environment for the Java-based OpenCCM implementation (included with the CADENA distribution) that includes code generation facilities for component implementation skeletons and CCM's configuration and deployment infrastructure.

In a similar fashion, we are building an end-to-end development environment for sensor network product lines with nesC as the underlying infrastructure using the collection of styles presented in the preceding sections. The current implementation includes nesC code import/export facilities that are able to process the entire component library provided with the nesC distribution. As an indicator of the number of artifacts, processing the primary *system* section of the library that is used in almost every application build yields 129 interface types, 97 component types, and 49 scenarios. Overall the library contains approximately 275 interface types and 456 component types. The models for these libraries will be included in the next CADENA release (expected May 2007).

Researchers at KSU's Sensor Network Center of Excellence are using this framework to develop product lines for multiple application domains including large livestock herd health monitoring, ground water run-off sensing, and radiation detection and response systems – these efforts are providing significant opportunities for us to evaluate modeling facilities presented here. CALM's ability to support multiple linked styles within a single modeling framework is playing a significant role in the design of the framework. For example, we are able to provide several higher-level modeling languages above the nesC style that aid scientists who are experts in the application domains (but not in the area of sensor networks) in carrying out the initial design of a sensor system.

7 Assessment

In this work, we have argued for an increased use of typing to specify and enforce structural constraints in modeling of component-based systems. A type-based approach provides a number of benefits and useful capabilities, it reduces the need to rely on more complicated forms of constraint checking, and it complements other forms of structural and behavioral constraint specification.

Space constraints make it impossible to present the details of complete framework, and we refer the reader to the CADENA website [15] for details. Website materials illustrate how we have validated the concepts presented here by giving a complete formalization of the typing system and by building a robust tool framework that has been used (a) to specify several widely used component models, and (b) to implement a complete development environment for a Java-based version of CCM. CADENA is being used by Lockheed Martin engineers to specify representative aspects of architectures for satellite mission control systems.[2]

[2]While non-disclosure agreements prevent us from reporting on the details of this work, the CADENA web-site contains PowerPoint slides from a two-hour demo given at Lockheed Martin STL in August 2006.

8 Logical Foundations for Conditional Information Flow Contracts

8.1 Background

National and international infrastructures as well as commercial services are increasingly relying on complex distributed systems that share information with *Multiple Levels of Security* (MLS). These systems often seek to coalesce information with mixed security levels into information streams that are targeted to particular clients. For example, in a national emergency response system, some data will be privileged (*e.g.*, information regarding availability of military assets, and deployment orders for those assets) and some data will be public (*e.g.*, weather and mapping information).

The *Multiple Independent Levels of Security* (MILS) architecture [81] proposes to make development, accreditation, and deployment of MLS-capable systems more practical, achievable, and affordable by providing a certified infrastructure *foundation for systems that require assured information sharing*. In the MILS architecture, systems are developed on top of: (a) a "separation kernel", a concept introduced by Rushby [74] which guarantees isolation and controlled communication between application components deployed in different virtual "partitions" supported by the kernel, and (b) MLS middleware services such as "high assurance guards" that allow information to flow between various partitions, and between trusted and untrusted segments of a network, only when certain *conditions* are satisfied.

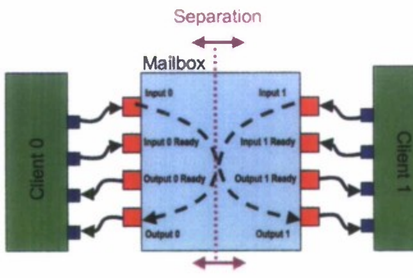
Researchers at the Rockwell Collins Advanced Technology Center are industry leaders in certifying MILS components according to standards such as the Common Criteria (EAL 6/7) that mandate the use of formal methods. For example, Rockwell Collins engineers carried out the certification of the hardware-based separation kernel in Rockwell Collins' AAMP7 processor (this was the first such certification of a MILS separation kernel and it formed the initial draft of the Common Criteria Protection Profile for Separation Kernels). Product groups at Rockwell Collins are building several different information assurance products on top of the AAMP7 that leverage the underlying MILS architecture. These products are often programmed using the SPARK subset of Ada [11]. A motivating factor for the use of SPARK is that it includes annotations (formal contracts for procedure interfaces) for specifying and checking information flow [13]. These annotations often play a key role in the certification of such products. The SPARK language and associated tool-set is the only commercial product that we know of which can support checking of code-level information flow contracts, and SPARK provides a number of well-designed and effective capabilities for specifying and verifying properties of highly critical implementations.

Even with SPARK, however, developers are sometimes unable to provide complete, machine-checkable arguments for the correctness of information assurance products. This is due to certain limitations in the SPARK information flow framework, in particular: SPARK information flow annotations are unconditional (e.g., they capture such statements as "executing procedure P may cause information to flow from input variable X to output variable Y"), but MLS security policies are often conditional (e.g., "data from input variable X is only allowed to flow to output variable Y when state variables G_1 and G_2 satisfy certain conditions"). Thus, SPARK currently can neither capture nor support verification of certain critical aspects of MLS security policies (treating such conditional flows as unconditional flows in SPARK is an over-approximation that leads to many false alarms).

In previous work, Amtoft and Banerjee have developed Hoare logics that enable compositional reasoning about information flow [5, 4]. Inspired by challenge problems from Rockwell Collins, these logics were extended to support conditional information flow [7]. While the logic as presented in [7] exposed some foundational issues, it only supported intraprocedural analysis, it required developers to specify information flow loop invariants, the verification algorithm was not yet fully implemented (and thus no experience was reported), and the core logic was not mapped to a practical method contract language capable of supporting compositional reasoning in industrial settings.

In this paper, we address these limitations by describing how the logic can provide a foundation for a practical information flow contract language capable of supporting compositional reasoning about conditional information flows. The specific contributions of our work are as follows:

- we propose an extension to SPARK's information flow contract language that supports conditional information flow, and we describe how the logic of [7] can be used to provide a semantics for the resulting framework,
- we extend the algorithm of [7] to support procedure calls and thus modularity,
- we present a strategy for automatically inferring conditional information flow invariants for while loops, thus significantly reducing developers' annotation burden,
- we provide an implementation that can automatically generate conditional information flow contracts from unannotated source code, and
- we report on experiments applying the implementation to a collection of examples.



```
procedure MACHINE_STEP
— INFORMATION FLOW CONTRACT (Figure 2)
is D_0, D_1 : CHARACTER;
begin
  if IN_0_RDY and not OUT_1_RDY then
    D_0 := IN_0_DAT;  IN_0_RDY := FALSE;
    OUT_1_DAT := D_0; OUT_1_RDY := TRUE;
  end if;
  if IN_1_RDY and not OUT_0_RDY then
    D_1 := IN_1_DAT;  IN_1_RDY := FALSE;
    OUT_0_DAT := D_1; OUT_0_RDY := TRUE;
  end if;
end MACHINE_STEP;
```

Figure 15: Simple MLS Guard - mailbox mediates communication between partitions.

Recent efforts for certifying MILS separation kernels [30, 35] applied ACL2 [42] or PVS [64] theorem provers to formal models; extensive inspections were then required by certification authorities to establish the correspondence between model and source code. Because our approach is directly integrated with code, it complements these earlier efforts by: (a) removing the "trust gaps" associated with inspecting behavioral models (built manually), and (b) allowing many verification obligations to be discharged earlier in the life cycle by developers while leaving only the most complicated obligations to certification teams. Moreover, our logic-based approach provides a foundation for producing independently auditable and machine-checkable *evidence* of correctness and MILS policy compliance as recommended [39] by the National Research Council's Committee on Certifiably Dependable Software Systems.

8.2 Example

Figure 15 illustrates the conceptual information flows in a fragment of a simplistic MLS component. Rockwell Collins engineers constructed this example to illustrate, to NSA and industry representatives, the specification and verification challenges facing the developers of MLS software. The "Mailbox" component in the center of the diagram mediates communication between two client processes – each running on its own partition in the separation kernel. *Client 0* writes data to communicate in the memory segment *Input 0* that is shared between *Client 0* and the mailbox, then it sets the *Input 0 Ready* flag. The mailbox process polls its ready flags; when it finds that, *e.g.*, *Input 0 Ready* is set and *Output 1 Ready* is cleared (indicating that *Client 1* has already consumed data deposited in the *Output 1* slot in a previous communication), then it copies the data from *Input 0* to *Output 1* and clears *Input 0 Ready* and sets *Output 1 Ready*. The communication from *Client 1* to *Client 0* follows a symmetric set of steps. The actions to be taken in each execution frame are encoded in SPARK by the `MACHINE_STEP` procedure of Fig. 15.

Figure 16(a) shows SPARK annotations for the `MACHINE_STEP` procedure, whose information flow properties are captured by `derives` annotations. It requires that each parameter and each global variable referenced by the procedure be classified as `in` (read only), `out` (written, and initial values [values at call point] are unread), or `in out` (written, and initial values read). For a procedure P, variables annotated as `in` or `in out` are called *input variables* and denoted IN_P; variables annotated as `out` or `in out` are *output variables* and denoted as OUT_P. Each output variable x_o must have a `derives` annotation indicating the input variables whose initial values are used to directly or indirectly calculate the final value of x_o. One can also think of each derives clause as expressing a dependence relation (or program slice) between an output variable and the input variables that it transitively depends on (via both data and control dependence). For example, the second `derives` clause specifies that on each `MACHINE_STEP` execution the output value of `OUT_1_DAT` is possibly determined by the input values of several variables: from `IN_0_DAT` when the Mailbox forwards data supplied by *Client 0*, from `OUT_1_DAT` when the conditions on the ready flags are not satisfied (`OUT_1_DAT`'s output value then is its input value), and from `OUT_1_RDY` and `IN_0_RDY` because these variables *control* whether or not data flows from *Client 0* on a particular machine step (*i.e.*, they *guard* the flow).

While upper levels of the MILS architecture require reasoning about lattices of security levels (*e.g.*, *unclassified*, *secret*, *top secret*), the policies of infrastructure components such as separation kernels and guard applications usually focus on data separation policies (reasoning about flows between components of program state), and we restrict ourselves to such reasoning in this paper.

No other commercial language framework provides automatically checkable information flow specifications, so the use of the information flow checking framework in SPARK is a significant step forward. As illustrated above,

```
--# global in out IN_0_RDY, IN_1_RDY,
--#                OUT_0_RDY, OUT_1_RDY,
--#                OUT_0_DAT, OUT_1_DAT;
--#         in     IN_0_DAT, IN_1_DAT;
--# derives
--#  OUT_0_DAT from IN_1_DAT, OUT_0_DAT,
--#                 OUT_0_RDY, IN_1_RDY &
--#  OUT_1_DAT from IN_0_DAT, OUT_1_DAT,
--#                 IN_0_RDY, OUT_1_RDY &
--#  IN_0_RDY  from IN_0_RDY, OUT_1_RDY &
--#  IN_1_RDY  from IN_1_RDY, OUT_0_RDY &
--#  OUT_0_RDY from OUT_0_RDY, IN_1_RDY &
--#  OUT_1_RDY from OUT_1_RDY, IN_0_RDY;
```

(a)

```
--# derives
--#  OUT_0_DAT from
--#     IN_1_DAT when
--#        (IN_1_RDY and not OUT_0_RDY),
--#     OUT_0_DAT when
--#        (not IN_1_RDY or OUT_0_RDY),
--#     OUT_0_RDY, IN_1_RDY &
--#  OUT_1_DAT from
--#     IN_0_DAT when
--#        (IN_0_RDY and not OUT_1_RDY),
--#     OUT_1_DAT when
--#        (not IN_0_RDY or OUT_1_RDY),
--#     OUT_1_RDY, IN_0_RDY
```

(b)

Figure 16: (a) SPARK information flow contract for Mailbox example. (b) Fragment of same example with proposed conditional information flow extensions (Section 8.4).

SPARK `derives` clauses can be used to specify flows of information from input variables to output variables, but they do not have enough expressive power to state that information only flows under specific conditions. For example, in the Mailbox code, information from `IN_0_DAT` only flows to `OUT_1_DAT` when the flag `IN_0_RDY` is set and the flag `OUT_1_READY` is cleared. Unfortunately, the SPARK `derives` cannot distinguish the flag variables as guards nor phrase the conditions under which the guards allow information to pass or be blocked. This means that guarding logic, which is central to many MLS applications including those developed at Rockwell Collins, *is completely absent from the checkable specifications* in SPARK. In general, the lack of ability to express *conditional* information flow not only inhibits automatic verification of guarding logic specifications, but also results in imprecision which cascades and builds throughout the specifications in the application.

8.3 Foundations of SPARK Conditional Information Flow

The SPARK subset of Ada is designed for programming and verifying high assurance applications such as avionics applications certified to DO-178B Level A. It deliberately omits constructs that are difficult to reason about such as dynamically created data, pointers, and exceptions. Below, we present the syntax of a simple imperative language with assertions that one can consider to be an idealized version of SPARK.

$$
\begin{array}{lll}
\textit{Assertions} & \textit{Expressions} & \textit{Commands} \\
\phi \ ::= \ B \mid \phi \wedge \phi & A \ ::= \ x \mid c \mid A \,\mathsf{op}\, A & S \ ::= \ \mathbf{skip} \mid x := A \mid \mathbf{assert}(\phi) \\
\quad\ \mid \ \phi \vee \phi \mid \neg\phi & B \ ::= \ A \,\mathsf{bop}\, A & \quad\ \mid \ S\,;S \mid \mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ S \\
 & & \quad\ \mid \ \mathbf{call}\ p \mid \mathbf{while}\ B\ \mathbf{do}\ S
\end{array}
$$

Features of SPARK that we do not consider here include the package and inheritance structure, records, and arrays. From these, only arrays present conceptual challenges. Our current implementation treats arrays as atomic entities, just as SPARK does. The extended version of this paper [8] describes how our logical approach can reason about individual elements of arrays (giving more precision than SPARK), a feature which is currently being included in our implementation. We consider both arithmetic (A) and boolean (B) expressions where we use $x, y, \ldots$ to range over variables, c to range over integer constants, p to range over named (parameterless) procedures, op to range over arithmetic operators in $\{+, \times, \mod, \ldots\}$, and bop to range over comparison operators in $\{=, <, \ldots\}$. Using parameterless procedures simplifies our exposition; our implementation supports procedures with parameters (there are no conceptual challenges in this extended functionality). For an expression E (arithmetic or boolean), we write $\mathsf{fv}(E)$ for the variables occurring free in E, and $E[A/x]$ for the result of substituting in E all occurrences of x by A.

The semantics of an arithmetic expression $[\![A]\!]$ is a function from stores into values, where a value ($v \in \mathit{Val}$) is an integer n and where a *store* $s \in \mathit{Store}$ maps variables to values; we write $dom(s)$ for the domain of s and write $[s \,|\, x \mapsto v]$ for the store that is like s except that it maps x into v. Similarly, $[\![B]\!]_s$ denotes a boolean. A command transforms the store into another store; hence its semantics is given in relational style, in the form $s[\![S]\!]s'$. For some S and s, there may not exist any s' such that $s[\![S]\!]s'$; this can happen if a **while** loop does not terminate, or an **assert** fails. The details of the semantics are standard and thus omitted; implicitly we assume a global procedure environment P that for each p returns a relation between input and output stores.

Assertions ϕ are also called 1-assertions since they represent predicates on a single program state; we write $s \models \phi$ to denote that ϕ holds in s following the standard semantics. We write $\phi \rhd_1 \phi'$ if whenever $s \models \phi$ also $s \models \phi'$. As

```
    {IN_1_RDY ∧ ¬OUT_0_RDY ⇒ IN_1_DAT⋉,
     ¬IN_1_RDY ∨ OUT_0_RDY ⇒ OUT_0_DAT⋉,
     IN_1_RDY⋉, OUT_0_RDY⋉}
1. if IN_1_RDY and not OUT_0_RDY then
    {IN_1_DAT⋉}
2.   D_1 := IN_1_DAT; IN_1_RDY := false;
    {D_1⋉}
3.   OUT_0_DAT := D_1; OUT_0_RDY := true;
    {OUT_0_DAT⋉}
4. fi
    {OUT_0_DAT⋉}
```

(a)

Summary information for p with $\mathrm{OUT}_p = \{x\}$

```
derives x from y,
               z when y > 0,
               w when y ≤ 0
```

Procedure call

```
{z > 7 ⇒ v⋉, z > 5 ⇒ u⋉, z > 5 ⇒ y⋉,
 z > 5 ∧ y > 0 ⇒ z⋉, z > 5 ∧ y ≤ 0 ⇒ w⋉}
  call p
{x > 5 ∧ z > 7 ⇒ v⋉,
 x > 7 ∧ z > 5 ⇒ (x + u)⋉}
```

(b)

Figure 17: (a) A derivation for the mailbox example, illustrating the handling of conditionals. (b) An example illustrating the handling of procedure calls (Section 8.5).

usual we define $\phi_1 \rightarrow \phi_2$ as $\neg\phi_1 \vee \phi_2$; we also define *true* as $0 = 0$, and *false* as $0 = 1$.

Reasoning about information flow in terms of non-interference: MILS seeks to prevent security breaches that can occur via unauthorized/unintended information flow from one partition to another; thus previous certification efforts for MILS components have among the core requirements included the classical property of *non-interference* [28] which (in this setting) states: for every pair of runs of a program, if the runs agree on the initial values of one partition's data (but may disagree on the data of other partitions) then the runs also agree on the final values of that partition's data.

Capturing non-interference and secure information flow in a compositional logic: The logic developed in [5] was designed to verify specifications of the following form: *given two runs of P that initially agree on variables $x_1 \ldots x_n$, the runs agree on variables $y_1 \ldots y_m$ at the end of the runs.* This includes non-interference as a special case (let $x_1 \ldots x_n$, and $y_1 \ldots y_m$, be the variables of one partition). We may express such a specification, which makes the "end-to-end" (input to output) aspect of verifying confidentiality explicit, in Hoare-logic style as $\{x_1 \ltimes, \ldots, x_n \ltimes\}\ P\ \{y_1 \ltimes, \ldots, y_m \ltimes\}$, where the *agreement assertion* $x\ltimes$ is satisfied by a *pair* of states, s_1 and s_2, if $s_1(x) = s_2(x)$. With P the example program from Sect. 8.2, we would have, e.g.,

$$\{\texttt{IN\_1\_DAT}\ltimes,\ \texttt{OUT\_0\_DAT}\ltimes,\ \texttt{IN\_1\_RDY}\ltimes,\ \texttt{OUT\_0\_RDY}\ltimes\}\ P\ \{\texttt{OUT\_0\_DAT}\ltimes\}.$$

To capture conditional information flow, recent work [7] by Banerjee and Amtoft introduced *conditional* agreement assertions, also called *2-assertions*. They are of the form $\phi \Rightarrow E\ltimes$ which is satisfied by a pair of stores if either at least one of them does not satisfy ϕ, or they agree on the value of E:

$$s\ \&\ s_1 \models \phi \Rightarrow E\ltimes \text{ iff whenever } s \models \phi \text{ and } s_1 \models \phi \text{ then } [\![E]\!]_s = [\![E]\!]_{s_1}.$$

We use $\theta \in \mathbf{2Assert}$ to range over 2-assertions. For $\theta = (\phi \Rightarrow E\ltimes)$, we call ϕ the antecedent of θ and write $\phi = ant(\theta)$, and we call E the consequent of θ and write $E = con(\theta)$. We often write $E\ltimes$ for *true* $\Rightarrow E\ltimes$. We use $\Theta \in \mathcal{P}(\mathbf{2Assert})$ to range over sets of 2-assertions (where we often write θ for the singleton set $\{\theta\}$), with conjunction implicit. Thus, $s\&s_1 \models \Theta$ iff $\forall\theta \in \Theta : s\&s_1 \models \theta$.

Fig. 17(a) illustrates a simple derivation using conditional information flow assertions that answers the question: what is the source of information flowing into variable `OUT_0_DAT`? The natural way to read the derivation is from the bottom up (since our algorithm works "backwards"). Thus, for `OUT_0_DAT`$\ltimes$ to hold after execution of P, we must have `D_1`$\ltimes$ before line 3 (since data flows from `D_1` to `OUT_0_DAT`), `IN_1_DAT`$\ltimes$ before line 2 (since data flows from `IN_1_DAT` to `D_1`), and before line 1 `IN_1_RDY`$\ltimes$ and `OUT_0_RDY`$\ltimes$ (since they *control* which branch of the condition is taken), along with conditional assertions. The pre-condition shows that the value of `OUT_0_DAT` depends *unconditionally* on `IN_1_RDY` and `OUT_0_RDY`, and *conditionally* on `IN_1_DAT` and `OUT_0_DAT`, just as we would expect.

Relations between agreement assertions: We define $\Theta \rhd_2 \Theta'$ to hold iff for all s, s_1: whenever $s\&s_1 \models \Theta$ then also $s\&s_1 \models \Theta'$. In development terms, when $\Theta \rhd_2 \Theta'$ holds we can think of Θ as a *refinement* of of Θ', and Θ' an *abstraction* of Θ. For example, $\{x\ltimes\ ,\ y\ltimes\}$ refines $x\ltimes$ by adding an (unconditional) agreement assertion, and $z < 10 \Rightarrow x\ltimes$ refines $z < 7 \Rightarrow x\ltimes$ by weakening the antecedent of a 2-assertion.

We define a function *decomp* that converts arbitrary 2-assertions into assertions with only variables as consequents: $decomp(\Theta) = \{\phi \Rightarrow x\ltimes \mid \phi \Rightarrow E\ltimes \in \Theta, x \in \mathrm{fv}(E)\}$. For example, $decomp(\phi \Rightarrow (x + y)\ltimes) = \{\phi \Rightarrow x\ltimes, \phi \Rightarrow y\ltimes\}$.

Fact 1 *For all Θ, $decomp(\Theta)$ is a refinement of Θ.*

The converse does not hold, with a counterexample being $s\&s_1 \models (x + y)\ltimes$ but not $s\&s_1 \models x\ltimes$ or $s\&s_1 \models y\ltimes$, as when $s(x) = s_1(y) = 3$, $s(y) = s_1(x) = 7$.

8.4 Conditional Information Flow Contracts

8.4.1 Foundations of flow contracts

The syntax of a SPARK `derives` annotation for a procedure P (as illustrated in Figure 16(a)) can be represented formally as a relation $\mathcal{D}_P$ between OUT_P and $\mathcal{P}(\text{IN}_P)$. A particular clause $\mathsf{derives}(x, \vec{y}) \in \mathcal{D}_P$ declares that the final value of output variable x depends on the input values of variables $\vec{y} = y_1, \ldots, y_k$. The correctness of such a clause as a contract for P can be expressed in terms of the logic of the preceding section, as requiring the triple $\{\vec{y}\ltimes\}\ S\ \{x\ltimes\}$ where S is the body of procedure P and where $\vec{y}\ltimes$ is a shorthand for $\{y_1\ltimes, \ldots, y_k\ltimes\}$.

Because $\mathcal{D}_P$ contains multiple clauses (one for each output variable of P), it captures multiple "channels" of information flow through P. Therefore, we cannot simply describe the semantics of a multi-clause derives contract $\{\mathsf{derives}(x, \vec{y}), \mathsf{derives}(z, \vec{w})\}$ as $\{(\vec{y}\vec{w})\ltimes\}\ S\ \{x\ltimes, z\ltimes\}$ because this would confuse the dependencies associated with x and z, *i.e.*, it would allow z to depend on $\vec{y}$. Accordingly, the full semantics of SPARK derives contracts is supported by what we term a *multi-channel version* of the logic which is extended to include *indexed agreement assertions* $x\ltimes_c$ indexed by a channel identifier c – which one can associate with a particular output variable. In the multi-channel logic, the confused triple above can now be correctly stated as $\{\vec{y}\ltimes_x, \vec{w}\ltimes_z\}\ S\ \{x\ltimes_x, z\ltimes_z\}$. (Alternatively, we could have *two* single-channel triples: $\{\vec{y}\ltimes\}\ S\ \{x\ltimes\}$ and $\{\vec{w}\ltimes\}\ S\ \{z\ltimes\}$.) The algorithm to be given in Sect. 8.5 extends to the multi-channel version of the logic in a straightforward manner, and our implementation supports the multi-channel version of the logic. For simplicity, we present the semantics of contracts using the single-channel version of the logic.

We now give a more convenient notation for triples of the form $\{\Theta\}P\{\Theta'\}$. A flow judgement κ is of the form $\Theta \leadsto \Theta'$, with Θ the precondition and with Θ' the postcondition. We say that $\Theta \leadsto \Theta'$ is valid for command S, written $S \models \Theta \leadsto \Theta'$, if whenever $s_1 \& s_2 \models \Theta$ and $s_1 [\![S]\!] s'_1$ and $s_2\ [\![S]\!]\ s'_2$ then also $s'_1 \& s'_2 \models \Theta'$ (if the 2-assertions in the precondition hold for input states s_1 and s_2, the postcondition must also hold for associated output states s'_1 and s'_2).

8.4.2 Language design for conditional SPARK contracts

The logic of the preceding section is potentially much more powerful than what we actually want to expose to developers – instead, we view it as a "core calculus" in which information flow reasoning is expressed. To determine how much of the power of the logic we wish to expose to developers in enhanced SPARK conditional information flow contracts, our design goals are: (1) writing the contracts should be as simple as possible, (2) the contracts should be able to capture common idioms of MILS information guarding, (3) the contract checking framework should be compositional so as to support MILS goals, and (4) there should be a natural progression (e.g., via formal refinements) from unconditional `derives` statements to conditional statements.

Simplifying assertions: The agreement assertions from the logic of Sect. 8.3 have the form $\phi \Rightarrow E\ltimes$. Here E is an arbitrary expression (not necessarily a variable), whereas SPARK `derives` statements are phrased in terms of IN/OUTvariables only. We believe that including arbitrary expressions in SPARK conditional `derives` statements would add significant complexity for developers, and our experimental studies have shown that little increase in precision would be gained by such an approach. Instead, we retain the use of expression-based assertions $\phi \Rightarrow E\ltimes$ only during intermediate (automated) steps of the analysis. Appealing to Fact 1, we have a canonical way of strengthening, at procedure boundaries, $\phi \Rightarrow E\ltimes$ to $\phi \Rightarrow w_1\ltimes, \ldots, \phi \Rightarrow w_k\ltimes$ where $\text{fv}(E) = \{w_1, \ldots, w_k\}$. A second simplification relates to the fact that the core logic allows both pre- and post-conditions to be conditional (e.g., $\{\phi_1 \Rightarrow E_1\ltimes\}\ P\ \{\phi_2 \Rightarrow E_2\ltimes\}$ where ϕ_1 and ϕ_2 may differ). Based on discussions with developers at Rockwell Collins and initial experiments, we believe that this would expose too much power/complexity to developers leading to unwieldy contracts and confusion about the underlying semantics. Accordingly, we are currently pursuing an approach in which only preconditions can be conditional. Combining these two simplifications, SPARK `derives` clauses are extended to allow conditions on input variables as follows:

$$\texttt{derives}\ x\ \texttt{from}\quad y_1\ \texttt{when}\ \phi_1, \quad \ldots, \quad y_k\ \texttt{when}\ \phi_k$$

Here $\phi_1 \ldots \phi_k$ are boolean expressions on the pre-state of the associated procedure P. Thus, the above specification can be read as "The value of variable x at the conclusion of executing P (for *any* final state s') is derived from those y_j where ϕ_j holds in the pre-state s from which s' is computed." Figure 16(b) shows how this can be used to specify conditional flows for procedure `MACHINE_STEP` in Fig. 15.

Design methodology separating guard logic from flow logic: The lack of conditional assertions in post-conditions has the potential to introduce imprecision. Yet, we believe the above approach to conditional expressions can be effective for the following reason: we have observed that information assurance application design tends to factor out the *guarding logic* (i.e., the pieces of state and associated state changes that determine *when* information can flow)

$\{\Theta\}\ (R) \Longleftarrow \textbf{skip}\ \{\Theta'\}$ iff $R = \{(\theta, u, \theta) \mid \theta \in \Theta'\}$ and $\Theta = \Theta'$

$\{\Theta\}\ (R) \Longleftarrow \textbf{assert}(\phi_0)\ \{\Theta'\}$ iff $R = \{((\phi \wedge \phi_0) \Rightarrow E\ltimes, u, \phi \Rightarrow E\ltimes) \mid \phi \Rightarrow E\ltimes \in \Theta'\}$ and $\Theta = dom(R)$

$\{\Theta\}\ (R) \Longleftarrow x := A\ \{\Theta'\}$ iff $R = \{(\phi[A/x] \Rightarrow E[A/x]\ltimes, \gamma, \phi \Rightarrow E\ltimes) \mid \phi \Rightarrow E\ltimes \in \Theta'\}$,
where $\gamma = m$ iff $x \in \text{fv}(E)$, and $\Theta = dom(R)$

$\{\Theta\}\ (R) \Longleftarrow S_1 ; S_2\ \{\Theta'\}$ iff $\{\Theta''\}\ (R_2) \Longleftarrow S_2\ \{\Theta'\}$ and $\{\Theta\}\ (R_1) \Longleftarrow S_1\ \{\Theta''\}$
and $R = \{(\theta, \gamma, \theta') \mid \exists \theta'', \gamma_1, \gamma_2 : (\theta, \gamma_1, \theta'') \in R_1,\ (\theta'', \gamma_2, \theta') \in R_2\}$, where $\gamma = m$ iff $\gamma_1 = m$ or $\gamma_2 = m$

$\{\Theta\}\ (R) \Longleftarrow \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{\Theta'\}$
iff $\{\Theta_1\}\ (R_1) \Longleftarrow S_1\ \{\Theta'\}$, $\{\Theta_2\}\ (R_2) \Longleftarrow S_2\ \{\Theta'\}$, $R = R'_1 \cup R'_2 \cup R'_0 \cup R_0$, and $\Theta = dom(R)$,
where $R'_1 = \{((\phi_1 \wedge B) \Rightarrow E_1\ltimes, m, \theta') \mid \theta' \in \Theta'_m,\ (\phi_1 \Rightarrow E_1\ltimes, \_, \theta') \in R_1\}$
and $R'_2 = \{((\phi_2 \wedge \neg B) \Rightarrow E_2\ltimes, m, \theta') \mid \theta' \in \Theta'_m,\ (\phi_2 \Rightarrow E_2\ltimes, \_, \theta') \in R_2\}$
and $R'_0 = \{(((\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)) \Rightarrow B\ltimes, m, \theta') \mid \theta' \in \Theta'_m, (\phi_1 \Rightarrow E_1\ltimes, \_, \theta') \in R_1, (\phi_2 \Rightarrow E_2\ltimes, \_, \theta') \in R_2\}$
and $R_0 = \{(((\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)) \Rightarrow E\ltimes, u, \theta') \mid \theta' \in \Theta'_u, (\phi_1 \Rightarrow E\ltimes, u, \theta') \in R_1, (\phi_2 \Rightarrow E\ltimes, u, \theta') \in R_2\}$
and $\Theta'_m = \{\theta' \in \Theta' \mid \exists(\_, m, \theta') \in R_1 \cup R_2\}$ and $\Theta'_u = \Theta' \setminus \Theta'_m$

$\{\Theta\}\ (R) \Longleftarrow \textbf{call}\ p\ \{\Theta'\}$
iff $R = R_u \cup R_0 \cup R_m$ and $\Theta = dom(R)$,
where $R_u = \{(rm^+_{\text{OUT}_P}(\phi) \Rightarrow E\ltimes, u, \phi \Rightarrow E\ltimes) \mid (\phi \Rightarrow E\ltimes) \in \Theta' \wedge \text{fv}(E) \cap \text{OUT}_P = \emptyset\}$
and $R_0 = \{(rm^+_{\text{OUT}_P}(\phi) \Rightarrow x\ltimes, m, \phi \Rightarrow E\ltimes) \mid (\phi \Rightarrow E\ltimes) \in \Theta' \wedge \text{fv}(E) \cap \text{OUT}_P \neq \emptyset \wedge x \in \text{fv}(E) \wedge x \notin \text{OUT}_P\}$
and $R_m = \{(rm^+_{\text{OUT}_P}(\phi) \wedge \phi^y_x \Rightarrow y\ltimes, m, \phi \Rightarrow E\ltimes)$
$\mid (\phi \Rightarrow E\ltimes) \in \Theta' \wedge x \in \text{fv}(E) \cap \text{OUT}_P \wedge \phi^y_x \Rightarrow y\ltimes$ among preconditions for $x\ltimes$ in p's summary $\}$

$\{\Theta\}\ (R) \Longleftarrow \textbf{while}\ B\ \textbf{do}\ S_0\ \{\Theta'\}$
iff $R = R_u \cup R_m$ and $\Theta = dom(R)$, where for each x (in X) we inductively in i define $\phi^i_x, \Theta^i, R^i, \psi^i_x$ by
$\phi^0_x = \bigvee\{\phi \mid \exists E : (\phi \Rightarrow E\ltimes) \in \Theta' \wedge x \in \text{fv}(E)\}$, $\quad \Theta^i = \{\phi^i_x \Rightarrow x\ltimes \mid x \in X\}$, $\quad \{\_\}\ (R^i) \Longleftarrow S_0\ \{\Theta^i\}$
$\psi^i_x = \bigvee\{\phi \mid \exists(\phi \Rightarrow E\ltimes, \_, \_) \in R^i,\ x \in \text{fv}(E)$ or $x \in \text{fv}(B),\ \exists(\theta, m, \theta') \in R^i,\ \phi \in \{ant(\theta), ant(\theta')\}\}$
$\phi^{i+1}_x =$ if $\psi^i_x \rhd_1 \phi^i_x$ then ϕ^i_x else $\phi^i_x \bigtriangledown \psi^i_x$,
and j is the least i such that $\Theta^i = \Theta^{i+1}$, and $R_m = \{(\theta, m, \theta') \mid \theta' \in \Theta'_m \wedge \theta \in \Theta^j \cup \{\mathit{true} \Rightarrow 0\ltimes\}\}$
and $R_u = \{(\phi \Rightarrow E\ltimes, u, \theta') \mid \theta' \in \Theta'_u, E = con(\theta'), (\text{fv}(E) = \emptyset, \phi = \mathit{true}) \vee (\text{fv}(E) \neq \emptyset, \phi = \bigvee_{x \in \text{fv}(E)}(\phi^j_x))\}$,
and $\Theta'_m = \{\theta' \in \Theta' \mid \exists x \in \text{fv}(con(\theta')) : \exists(\_, m, \_ \Rightarrow x\ltimes) \in R^j\}$ and $\Theta'_u = \Theta' \setminus \Theta'_m$

Figure 18: The Precondition Generator

from the code which propagates information. This follows a common pattern in embedded systems in which the control logic is often factored out from data computation logic.

Contract abstraction and refinement: For a practical design and development methodology, it is important to consider notions of contract abstraction (generalization) and refinement – ideally, conditional contracts should be a refinement of unconditional contracts. For example, we believe it will be easier to introduce conditional contracts into workflows if developers can: (1) make a rough cut at specifying information flows without conditions, and (2) systematically refine to produce conditional contracts, perhaps assisted by expert verification engineers. Conversely, if developers decide not to pursue a verification approach based on our conditional contracts, we want them to be able to safely abstract all conditional contracts back to unconditional SPARK contracts.

We now establish the desired notion of contract refinement (in terms of the general underlying calculus instead of its limited exposure in SPARK), by defining a relation between flow judgements: $\kappa_1 \rhd_\kappa \kappa_2$, pronounced "$\kappa_1$ refines κ_2", to hold iff for all commands S, whenever $S \models \kappa_1$ then also $S \models \kappa_2$. To gain the proper intuition about contract refinement, it is important to note that the refinement relation is contra-variant in the pre-condition and co-variant in the post-condition: given $\kappa_1 \equiv \Theta_1 \rightsquigarrow \Theta'_1$ and $\kappa_2 \equiv \Theta_2 \rightsquigarrow \Theta'_2$, if $\Theta_2 \rhd_2 \Theta_1$ and $\Theta'_1 \rhd_2 \Theta'_2$ then $\kappa_1 \rhd_\kappa \kappa_2$. For example, $x\ltimes \rightsquigarrow y\ltimes \rhd_\kappa x\ltimes, y\ltimes \rightsquigarrow y\ltimes$ holds because $x\ltimes, y\ltimes \rhd_2 x\ltimes$ (Section 8.3). Intuitively, this captures the fact that a contract can always be *abstracted* to a weaker one by stating that the output variables may depend on additional input variables. This illustrates that our contracts capture "may" dependence modalities: output y *may* depend on both inputs x and y, but a refinement $x\ltimes \rightsquigarrow y\ltimes$ shows that output y need not depend on input y (the contract before refinement is an *over-approximation* of dependence information). Also, we have $(z < 7 \Rightarrow x\ltimes \rightsquigarrow y\ltimes) \rhd_\kappa (x\ltimes \rightsquigarrow y\ltimes)$ which realizes our design goals of achieving: (a) a formal refinement by adding conditions to a contract, and (b) a formal (safe) abstraction by removing conditions.

8.5 A Precondition Generation Algorithm

We define in Fig. 18 an algorithm Pre for inferring preconditions from postconditions. We write $\{\Theta\}\ (R) \Longleftarrow S\ \{\Theta'\}$ when, given command S and postcondition Θ', Pre returns a precondition Θ for S that is designed so as to be sufficient to establish Θ', and a relation R that associates each 2-assertion $\theta \in \Theta'$ with the 2-assertions in Θ needed to establish θ. R captures dependences between variables before and after the execution of S, and it also supports reasoning about multiple channels of information flow as discussed in Sect. 8.4.1, *e.g.*, if $\{y_1 y_2 \ltimes_x, y_1 y_3 \ltimes_z\}\ S\ \{x\ltimes_x, z\ltimes_z\}$ then R

will relate y_1 to x and to z, y_2 to x, and y_3 to z. More precisely, we have $R \subseteq \Theta \times \{m, u\} \times \Theta'$ where tags m,u are mnemonics for "modified" and "unmodified"; if $(\theta, u, \theta') \in R$ then additionally it holds that S modifies no "relevant" variable, where a "relevant" variable is one occurring in the consequent of θ'. We use γ to range over $\{m, u\}$, and write $dom(R) = \{\theta \mid \exists(\theta, \_, \_) \in R\}$ and $ran(R) = \{\theta' \mid \exists(\_, \_, \theta') \in R\}$.

Correctness results: If $\{\Theta\}\ (\_) \Longleftarrow S\ \{\Theta'\}$ then Θ is indeed a precondition (but not necessarily the *weakest* such) that is strong enough to establish Θ', as stated by:

Theorem 8.1 (Correctness) *Assume* $\{\Theta\}\ (\_) \Longleftarrow S\ \{\Theta'\}$. *Then* $S \models \Theta \rightsquigarrow \Theta'$. *That is, if* $s\&s_1 \models \Theta$, *and* s', s'_1 *are such that* $s\ [\![S]\!]\ s'$ *and* $s_1\ [\![S]\!]\ s'_1$, *then* $s'\&s'_1 \models \Theta'$.

Note that Theorem 8.1 is termination-*in*sensitive; this is not surprising given our choice of a relational semantics (but see [6] for a logic-based approach that is termination-sensitive). Also note that correctness is phrased directly wrt. the underlying semantics, unlike [5, 4] which first establish the semantic soundness of a logic and next provide a sound implementation of that logic. Theorem 8.1 is proved [8] much as the corresponding result [7] (that handled a language with heap manipulation but without procedure calls and without automatic computation of loop invariants), by establishing some auxiliary properties (e.g., the R component) that have largely determined the design of Pre.

Intraprocedural analysis: We now explain the various clauses of Pre in Fig. 18, where the clause for **skip** is trivial. For an assignment $x := A$, each 2-assertion $\phi \Rightarrow E\ltimes$ in Θ' produces exactly one 2-assertion in Θ, given by substituting A for x (as in standard Hoare logic) in ϕ as well as in E; the connection is tagged m when x occurs in E. For example, if S is $x := w$ then R might contain the triplets $(q > 4 \Rightarrow w\ltimes, m, q > 4 \Rightarrow x\ltimes)$ and $(w > 3 \Rightarrow z\ltimes, u, x > 3 \Rightarrow z\ltimes)$. The rule for $S_1 ; S_2$ works backwards, first computing S_2's precondition which is then used to compute S_1's; the tags express that a consequent is modified iff it has been modified in either S_1 or S_2. The rule for **assert** allows us to weaken 2-assertions, by strengthening their antecedents; this is sound since execution will abort from stores not satisfying the new antecedents.

To illustrate and motivate the rule for conditionals, we shall use Fig. 17(a) where, given postcondition `OUT_0_DAT`$\ltimes$, the **then** branch generates (as the domain of R_1) precondition `IN_1_DAT`$\ltimes$ which by R'_1 contributes the first conditional assertion of the overall precondition. The **skip** command in the implicit **else** branch generates (as the domain of R_2) precondition `OUT_0_DAT`$\ltimes$ which by R'_2 contributes the second conditional assertion of the overall precondition. We must also capture that two runs, in order to agree on `OUT_0_DAT` after the conditional, must agree on the value of the test B; this is done by R'_0 which generates the precondition $(\mathit{true} \wedge B) \vee (\mathit{true} \wedge \neg B) \Rightarrow B\ltimes$; optimizations (not shown) in our algorithm simplify this to $B\ltimes$ and then use Fact. 1 to split out the variables in the conjuncts of B into the two unconditional assertions of the overall precondition. Finally, assume the postcondition contained an assertion $\phi \Rightarrow E\ltimes$ where E is not modified by either branch: if also ϕ is not modified then $\phi \Rightarrow E\ltimes$ belongs to both R_1 and R_2, and hence by R_0 also to the overall precondition; if ϕ is modified by one or both branches, R_0 generates a more complex antecedent for $E\ltimes$.

Interprocedural analysis: Recall from Sect. 8.4.2 that for a procedure summary, we allow only variables as consequents, and allow conditional assertions only in the preconditions. At a call site **call** p, antecedents in the call's postcondition will carry over to the precondition, *provided* that they do not involve variables in OUT$_P$. Otherwise, since our summaries express variable dependencies but not functional relationships, we cannot state an exact formula for modifying antecedents (unlike what is the case for assignments). Instead, we must conservatively strengthen the preconditions, by weakening their antecedents; this is done by an operator rm^+ such that if $\phi' = rm^+_X(\phi)$ (where $X = \text{OUT}_p$) then ϕ logically implies ϕ' where ϕ' does not contain any variables from X. A trivial definition of rm^+ is to let it always return *true* (which drops all conditions associated with X), but we can often get something more precise; for instance, we can choose $rm^+_{\{x\}}(x > 7 \wedge z > 5) = (z > 5)$.

Equipped with rm^+, we can now define the analysis of procedure call, as done in Fig. 18 and illustrated in Fig. 17(b). In Fig. 18, R_u deals with assertions (such as $x > 5 \wedge z > 7 \Rightarrow v\ltimes$ in the example) whose consequent has not been modified by the procedure call (its "frame conditions" determined by the OUT declaration). For an assertion whose consequent E has been modified (such as $x > 7 \wedge z > 5 \Rightarrow (x + u)\ltimes$), we must ensure that the variables of E agree after the procedure call (when the antecedent holds). For those not in OUT$_p$ (such as u), this is done by R_0 (which expresses some "semi frame conditions"); for those in OUT$_p$ (such as x), this is done by R_m which utilizes the procedure summary (contract) of the called procedure.

Synthesizing loop invariants: For while loops (the only iterative construct), the idea is to consider assertions of the form $\phi_x \Rightarrow x\ltimes$ and then repeatedly analyze the loop body so as to iteratively weaken the antecedents until a fixed point is reached. Illustratively:

Package.Procedure Name	LoC	C	L	P	O	SF	Flows		Cond. Flows		Gens.		Time (seconds)	
							1	2	1	2	1	2	1	2
Autopilot.AP.Altitude.Pitch.Rate.History_Average	10	0	1	0	1	2	5	3	0	0	0	0	0.047	0.063
Autopilot.AP.Altitude.Pitch.Rate.Calc_Pitchrate	13	2	0	2	2	7	17	8	0	0	15	15	0.000	0.015
Autopilot.AP.Altitude.Pitch.Target_Rate	17	4	0	1	1	3	53	4	42	0	142	46	0.015	0.015
Autopilot.AP.Heading.Roll.Target_ROR	15	3	0	1	1	2	4	3	0	0	26	26	0.000	0.000
Autopilot.AP.Heading.Roll.Target_Rate	11	2	0	1	1	3	9	4	0	0	14	14	0.000	0.000
Autopilot.AP.Control	19	1	0	13	8	46	58	54	0	0	63	51	0.016	0.032
Autopilot.Scale.Scale_Movement	22	4	0	2	1	4	47	10	46	9	0	0	0.016	0.000
Minepump.Logbuffer.ProtectedWrite	8	1	0	0	5	9	9	9	4	4	0	0	0.031	0.047
Mailbox.MACHINE_STEP	17	2	0	0	6	16	18	18	12	12	0	0	0.047	0.062
Mailbox.Main	6	0	1	1	6	16	54	22	0	0	2	2	0.031	0.016
BoilerWater-Monitor.FaultIntegrator.Test	11	3	0	0	4	11	46	22	42	18	0	0	0.047	0.047
BoilerWater-Monitor.FaultIntegrator.Main	11	0	1	6	2	2	14	4	0	0	0	0	0.016	0.016
Lift-Controller.Poll	22	2	1	3	2	9	77	12	43	0	0	0	0.031	0.031
Lift-Controller.Traverse	18	0	1	11	3	10	210	13	66	0	0	0	0.281	0.063
Missile_Guidance.Clock_Read	12	2	0	0	3	5	13	11	10	8	0	0	0.047	0.047
Missile_Guidance.Extrapolate_Speed	13	2	0	2	2	7	14	10	6	4	36	16	0.000	0.000
Missile_Guidance.Code_To_State	12	3	0	0	1	7	15	9	14	8	0	0	0.000	0.000
Missile_Guidance.Transition	20	4	0	2	1	9	3527	63	3524	62	4	4	0.156	0.125
Missile_Guidance.Drag_cfg.Calc_Drag	21	4	0	1	1	3	37	3	34	0	0	0	0.000	0.000
Missile_Guidance.Nav.Handle_Airspeed	18	4	0	4	3	13	117	28	110	25	18	18	0.000	0.000
Missile_Guidance.Nav.Estimate_Height	21	5	0	2	2	11	60	18	57	16	4	4	0.000	0.000

Table 1: Experiment Data (excerpts)

```
while i < 7 do
  if odd(i)
   then r := r + v; v := v + h
   else v := x;
  i := i + 1
{r⋉}
```

Iteration	0	1	2	3		
	false	*false*	*false*	*false*	$\Rightarrow$	$h\ltimes$
	false	*true*	*true*	*true*	$\Rightarrow$	$i\ltimes$
	true	*true*	*true*	*true*	$\Rightarrow$	$r\ltimes$
	false	$\mathsf{odd}(i)$	$\mathsf{odd}(i)$	$\mathsf{odd}(i)$	$\Rightarrow$	$v\ltimes$
	false	*false*	$\neg\mathsf{odd}(i)$	*true*	$\Rightarrow$	$x\ltimes$

Here we are given $r\ltimes$ as postcondition; hence the initial value of r's antecedent is *true* whereas all other antecedents are initialized to *false*. The first iteration updates v's antecedent to $\mathsf{odd}(i)$ (we use $\mathsf{odd}(i)$ as a shorthand for $i \mod 2 = 1$), since v is used to compute r when i is odd, and also updates i's antecedent to *true*, since (the parity of) i is used to decide whether r is updated or not. The second iteration updates x's antecedent to $\neg\mathsf{odd}(i)$, since in order for two runs to agree on v when i is odd, they must have agreed on x in the previous iteration when i was even. The third iteration updates x's antecedent to *true*, since in order for two runs to agree on x when i is even, then must agree on x always (as x doesn't change). We have now reached a fixed point. It is noteworthy that even though the postcondition mentions $r\ltimes$, and r is updated using v which in turn is updated using h, the generated precondition does not mention h, since the parity of i was exploited. This shows [7] that even if we should only aim at producing contracts where all assertions are unconditional, precision may still be improved if the analysis engine makes internal use of *conditional* assertions.

In the general case, however, fixed point iteration may not terminate. To ensure termination, we need a "widening operator" $\triangledown$ on 1-assertions, with the following properties: *(a)* for all ϕ and ψ, ψ logically implies $\psi \triangledown \phi$, and also ϕ logically implies $\psi \triangledown \phi$; *(b)* if for all i we have that ϕ^{i+1} is of the form $\phi^i \triangledown \psi$, then the chain $\{\phi^i \mid i \geq 0\}$ eventually stabilizes. A trivial widening operator is the one that always returns *true*, in effect converting conditional agreement assertions into unconditional. A less trivial option will utilize a number of assertions, say $\psi_1 \ldots \psi_k$, and allow $\psi \triangledown \phi = \psi_j$ if ψ_j is logically implied by ψ as well as by ϕ; such assertions may be given by the user if he has a hint that a suitable invariant may have one of $\psi_1 \ldots \psi_k$ as antecedent.

8.6 Evaluation

The algorithm of Section 8.5 provides a foundation for automatically *inferring* contracts from implementations, but can also be used for *checking* `derives` contracts supplied by a developer: the verification condition will be that the contract pre-condition implies the inferred pre-condition. In principle, this approach may reject a sound contract since the inference algorithm does not always generate the weakest pre-condition.

There is much merit in a methodology that encourages writing of the contract *before* writing/checking the implementation. However, one of our strategies for injecting our techniques into industrial development groups is to pitch the tools as being able to discover more precise conditional specifications to supplement conventional SPARK `derives` contracts already in the code; thus we focus the experimental studies of this section on the more challenging problem of automatically inferring *conditional* contracts starting from code with no existing `derives` annotations.

For each procedure P with $\mathrm{OUT}_P = \{w_1, \ldots, w_k\}$, the algorithm analyzes the body wrt. a post-condition $w_1 \ltimes_1, \ldots, w_k \ltimes_k$. Since SPARK disallows recursion, we simply move in a bottom-up fashion through the call-graph – guaranteeing that a contract exists for each called procedure. When deployed in actual development, one would probably allow developers to tweak the generated contracts (e.g., by removing unnecessary conditions for establishing end-to-end policies) before proceeding with contract inference for methods in the next level of the call hierarchy. However, in our experiments, we used autogenerated contracts for called methods without modification. All experiments were run under JDK 1.6 on a 2.2 GHz Intel Core2 Duo.

Code bases: Embedded security devices are the initial target domain for our work, and the security-critical sections to be certified from these code bases are often relatively small, *e.g.*, roughly 1000 LOC for one Rockwell Collins high assurance product and 3000 LOC for a device recently certified by Naval Research Labs researchers [35]. For our evaluation, we consider a collection of five small to moderate size applications from the SPARK distribution, in addition to an expanded version of the mailbox example of Section 8.2. Of these, the *Autopilot* and *Missile Control* applications are the most realistic. There are well over 250 procedures in the code bases, but due to space constraints, in Table 1 we list metrics for only the most complex procedures from each application (see [86] for the source code of all the examples). Columns **LOC**, **C**, **L**, and **P** report the number of non-comment lines of code, conditional expressions, loops, and procedure calls in each method. Our tool can run in two modes. The first mode (identified as version **1** in Table 1) implements the rules of Figure 18 directly, with just one small optimization: a collection of boolean simplifications are introduced, e.g., simplifying assertions of the form $\mathit{true} \wedge \phi \Rightarrow E\ltimes$ to $\phi \Rightarrow E\ltimes$. The second mode (version **2** in Table 1) enables a collection of simplifications aimed at compacting and eliminating redundant flows from the generated set of assertions. One simplification performed is elimination of assertions with *false* in the antecedent (these are trivially true), and elimination of duplicate assertions. Also, it eliminates simple entailed assertions, such as $\phi \Rightarrow E\ltimes$ when $\mathit{true} \Rightarrow E\ltimes$ also appears in the assertion set.

Typical refinement power of the algorithm: Column **O** gives the number of OUT variables of a procedure (this is equal to the number of `derives` clauses in the original SPARK contract), and Column **SF** gives the number of *flows* (total number of IN/OUT pairs) appearing in the original contract. Column **Flows** gives the number of flows generated by different versions of our algorithm. This number increases over **SF** as SPARK flows are refined into conditional flows (often creating two or more conditioned flows for a particular IN/OUT variable pair). The data shows that the compacting optimizations often substantially reduce the number of flows; the practical impact of this is to substantially increase the readability/tractability of the contracts. Column **Cond. Flows** indicates the number of flows from **Flows** that are conditional. We expect to see the refining power of our approach in procedures with conditionals (column **C**) primarily, but we also see increases in precision that is due to conditional contracts of called procedures (column **P**). In a few cases we see a blow-up in the number of conditional flows. The worse case is `MissileGuidance.Transition`, which contains a case statement with each branch containing nested conditionals and procedure calls with conditional contracts – leading to an exponential explosion in path conditions. Only a few variables in these conditions lie in what we consider to be the "control logic" of the system. The tractability of this example would improve significantly with the methodology suggested earlier in which developers declare explicitly the guarding variables (such as `IN_1_RDY` of Fig. 15), thus allowing the algorithm to omit tracking of conditional flows not associated with declared guard variables. A manual inspection of each inferred contract showed that the algorithm usually produces conditions that an expert would expect.

Efficiency of inference algorithm: As can be see in the **Time** columns, the algorithm is quite fast for all the examples, usually taking a little longer in version **2** (all optimizations on). However, for some examples, version **2** is actually faster; these are the cases of procedures with calls to other procedures. Due to the optimizations, the callees now have simpler contracts, simplifying the processing of the caller procedures.

Sources of loss of precision: We would like to determine situations where our treatment of loops or procedure calls leads to abstraction steps that discard conditional information. While this is difficult to determine for loops (one would have to compare to the most precise loop invariant – which would need to be written by hand), Column **Gens.** indicates the number of conditions dropped across processing of procedure calls. The data shows, and our experience confirms, that the loss of precision is not drastic (in some cases, one wants conditions to be discarded), but more experience is needed to determine the practical impact on verification of end-to-end properties.

8.7 Related Work

The theoretical framework for the SPARK information flow framework is provided by Bergeretti and Carré [13] who presents a compositional method for inferring and checking dependences among variables. That approach is flow-sensitive, whereas most security type systems [82, 10] are flow-*in*sensitive as they rely on assigning a security level ("high" or "low") to each variable. Chapman and Hilton [16] describe how SPARK information flow contracts could be extended with lattices of security levels and how the SPARK Examiner could be enhanced to check conformance of flows to particular security levels. Those ideas could be applied directly to provide security levels of flows in our framework. Rossebo *et al.*[72] show how the existing SPARK framework can be applied to verify various *unconditional* properties of a MILS Message Router. Apart from Spark Ada, there exists several tools for analyzing information flow properties, notably Jif (Java + information flow) which is based on [58]), and Flow Caml [76].

The seminal work on agreement assertions is [5], whose logic is flow-sensitive, and comes with an algorithm for computing (weakest) preconditions, but the approach does not integrate with programmer assertions. To address that, and to analyze heap-manipulating languages, the logic of [4] employs *three* kinds of primitive assertions: agreement, programmer, and region (for a simple alias analysis). But, since those can be combined only through conjunction, programmer assertions are not smoothly integrated, and it is not possible to capture *conditional* information flows. That was what motivated Amtoft & Banerjee [7] to introduce conditional agreement assertions, for a heap-manipulating language. This paper integrates that approach into the SPARK setting (whose lack of heap objects allows us to omit the "object flow invariants" of [7]) for practical industrial development, adds interprocedural contract-based compositional checking, adds an algorithm for computing loop invariants (rather than assuming the user provides them), and provides an implementation as well as reports on experiments.

A recently popular approach to information flow analysis is *self-composition*, first proposed by Barthe et al. [12] and later extended by, e.g., Terauchi and Aiken [78] and (for heap-manipulating programs) Naumann [61]. Self-composition works as follows: for a given program S, a copy S' is created with all variables renamed (primed); with the observable variables say x, y, then non-interference holds provided the sequential composition $S; S'$ when given precondition $x = x' \land y = y'$ also ensures postcondition $x = x' \land y = y'$. This is a property that can be checked using existing static verifiers.

Darvas et al. [20] use the KeY tool for interactive verification of non-interference; information flow is modeled by a dynamic logic formula, rather than by assertions.

When it comes to *conditional* information flow, the most noteworthy existing tool is the slicer by Snelting et al [77] which generates *path conditions* in program dependence graphs for reasoning about end-to-end flows between specified program points/variables. In contrast, we provide a contract-based approach for *compositional* reasoning about conditions on flows with an underlying logic representation that can provide external evidence for conformance to conditional flow properties. We have recently received the implementation of the approach in [77], and we are currently investigating the deeper technical connections between the two approaches.

Finally, we have already noted how our work has been inspired by and aims to complement previous ground-breaking efforts in certification of MILS infrastructure [30, 35]. While the direct theorem-proving approach followed in these efforts enables proofs of very strong properties beyond what our framework can currently handle, our aim is to dramatically reduce the labor required, and the potential for error, by integrating automated techniques directly on code, models, and developer workflows to allow many information flow verification obligations to be discharged earlier in the life cycle.

8.8 Assessment

We have presented what we believe to be an effective and developer-friendly framework for specification and automatic checking of conditional information flow properties, which are central to verification and certification of information applications hoping to provide MLS solutions. The directions that we are pursuing are inspired directly by challenge problems presented to us by industry teams using SPARK to develop MLS components. The initial prototyping and evaluation of our framework has produced promising results, and we are pressing ahead with evaluating our techniques against actual product codebases developed at Rockwell Colins. A crucial concern in this effort will be to develop design and implementation methodologies for (a) exposing and checking conditional information flows, (b) specifying and checking security levels of data along conditional flows, and (c) investigating a more precise treatment of arrays as presented in our technical report [8].

9 Browsing Java Dependences and Information Flow with Indus

9.1 Background

9.1.1 Slicing – Concepts and Applications

Program slicing is a well-known program analysis and transformation technique that uses program statement dependence information to identify parts of a program that influence or are influenced by an initial set of program points of interest (called the *slice criteria*). For instance, given a slicing criteria C consisting of a set of program statements, a program slicer computes a *backward slice* S_b containing all program statements that influence the statements in C by starting from C and successively adding to S_b statements upon which the C statements are (transitively) data or control dependent. A *forward slice* S_f containing all program statements that C influences is calculated in an analogous manner: the slicer successively adds to S_f all statements that are (transitively) data or control dependent on the statements in C. Upon conclusion of the slice calculation, the slicer may have the capability to (a) generate an *executable* slice — a residual program containing only the statements in the slice (perhaps with a few additional statements to guarantee well-formedness), or (b) to display the original program with nodes in the slice visually high-lighted in some way.

Slicing has been widely applied in the context of debugging, program comprehension, and testing.

- **Debugging:** When debugging software, it is often the case that a bug is detected at a state associated with single program point p_b (e.g., an assertion violation). If the software is large and complex, then it is likely that the software fault occurs at a program point p_f that is statically distant (i.e., in the source code) from the program point p_b. In such cases, the developer will need to methodically sift through the source code of the software to identify the faulting program point p_f. To expedite this process, the developer will attempt to limit the search to the parts of the software that may either directly or indirectly affect the behavior (state) of the program at the program point p_b. This process can be automated using backwards program slicing starting with p_b as the slicing criteria.

- **Program comprehension:** Software developers are frequently assigned to debug, further develop, or reverse engineer code bases that they did not author. In such cases, it is often difficult for the developer to grasp the basic architecture and relationships between code units, and this is made more difficult by the fact that the code may be poorly documented and poorly written. Both backward and forward slicing can be applied to browse the code, looking for dependences between code units, flows of data between program statements, etc.

- **Testing:** There are a number of applications of slicing in the context of testing. One particular example is *impact analysis* [71], which aims to determine the set of program statements or test cases that are impacted by a change in the program, requirements, or tests. For example, in verification and validation efforts on large code bases with huge test suites, it is often very expensive to run all the tests associated with the program. If a program statement p_b is modified (e.g., due to a bug fix), rather than re-running all tests, backwards slicing using p_b as the criteria can be used to determine the subset of the tests that actually influence the behavior of the program at the point of the bug fix, and only those relevant tests need to be re-run. In addition, a developer may want to understand the potential impact that the change at p_b can have on other statements of the program. Forward slicing with p_b as the criteria can be used to locate other statements within the program that will be impacted by the change at p_b.

9.1.2 Tool Support

There have been a large number of publications on slicing, but only a small number of implementations for languages such as FORTRAN, ANSI C, and Oberon.[3] Most of the implementations have been targeted to particular applications of program slicing such as program comprehension, testing, program verification, etc. Moreover, although slicing tools have been developed for programming languages like C, only a few slicing tools exist for languages like Java and C++ [60, 45].

Dealing with widely-used languages like Java, C++, C# involves a number of challenges.

- **Dealing with references and aliasing:** Calculation of data dependences (determining which definitions of a variable v reach a particular use of v) is made much more difficult by pointers/references and aliasing. It

[3]Please refer to Jens Krinke's Dissertation [45] for a brief informative overview of available implementations.

is difficult to determine statically which memory cells a variable of reference type may be pointing to, and sophisticated static analyses must be used to collect information about the memory cells that could possibly be referred to by a particular variable. For soundness, such analyses must be conservative (i.e., they must overestimate the set of cells that could be pointed to), and this approximating effect leads to imprecision in slicing (slices are larger than actually required for correctness).

- **Dealing with exceptions:** Modern languages like Java and C# support exception processing. The use of exceptions and associated exception handlers introduces implicit less-structured control flow into the program which makes it more difficult to calculate the *control dependence* information needed in slicing.
- **Dealing with concurrency:** The increasing use of multi-threading further hampers analysis since languages that emphasize a shared memory model (like Java and C#) allow accesses of a memory cell in one thread to be potentially interfering with accesses in another thread (thus, creating additional and often spurious program dependences). Reducing spurious dependences by determining that accesses do not actually interfere (e.g., as guaranteed through the use of proper locking or use of heap data that is actually not shared between threads) requires sophisticated static analyses that can detect lock states, situations where objects do not escape a particular thread context, and partial order information (e.g., detecting that actions of two different threads cannot interfere because one must definitely happen before the other).
- **Dealing with libraries:** Realistic programs make extensive use of libraries to the extent that a large majority of executable code comes from libraries as opposed to actual application code written by the developer. Slicing must be able to include program representations of relevant library code while excluding library code not actually invoked by the application code.

In summary, while the basic theory of slicing for a simple imperative language can be explained rather succinctly, building a robust tool environment for slicing realistic programs written in a language like Java requires both foundational work along a number of fronts as well as a large-scale tool engineering effort.

9.1.3 Motivation

Our work focuses on slicing realistic Java programs. We were originally motivated to build a slicer for Java because we were seeking ways to reduce the cost of model checking concurrent Java programs in the Bandera project [19][4]. Model checking is a verification and bug-finding technique that aims to perform an exhaustive exploration of a program's state space. In simple terms, model checking a concurrent Java program involves simulating all possible executions of the program (e.g., including all possible thread schedules) and checking the paths and states encountered in that simulation against correctness specifications phrased as assertions, automata, or temporal logic formulae. While model checking can be very effective for detecting intricate flaws that are hard to detect using conventional non-exhaustive techniques like testing, it is very expensive to apply. Thus, effective use of model checking must rely on applying different abstraction techniques, imposing bounds on the state space explored, and employing heuristics for state-space search.

The effectiveness of slicing for model reduction is based on the observation that, when trying to verify a particular specification ϕ against a program P, many parts of P do not impact whether ϕ ultimately holds for P or not. For example, it is often the case that ϕ is a simple assertion or a temporal property that only mentions a few of P's features (e.g., a few variable names or program points). Thus, one can use the features mentioned in ϕ to create a slice of P that omits program statements and variables that are irrelevant to ϕ's satisfaction against P. We have shown that using slicing in this manner forms a sound and complete reduction technique for model checking [34]. Our experimental studies on small to moderate size concurrent Java programs shows that slicing almost always provides some reduction (in best cases, up to a factor of four reduction in time), and incurs very little overhead compared to the end-to-end costs of model checking [24].

As we started our work, no slicing infrastructure for Java was available, and little technical work had been done to address challenges associated with slicing realistic programs mentioned above. Thus, following the maxim *"Every good work of software starts by scratching a developer's personal itch."*[5], the lack of a robust, flexible, and publicly available program slicer for concurrent Java motivated us to implement one ourselves.

[4]This software is available at http://bandera.projects.cis.ksu.edu.

[5]Cited from the paper titled "The Cathedral and the Bazaar" by Eric S. Raymond and available at http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

```
1 int bar(int k) {
2   int v;
3   if (k == 0)
4     v = 1;
5   else
6     v = 2;
7   return v;
8 }
```

```
1  int foo(int k) {
2    Pointer v, u;
3    v = new Pointer();
4    u = v;
5    if (k == 0)
6      v.o = 1;
7    else
8      v.o = 2;
9    u.o = 4;
10   return v.o;
11 }
```

Figure 19: Java examples to illustrate data and control dependence. `Pointer` is a class with an public integer field named `o`.

9.2 Program Slicing

Given a program P and a slice criteria C, one can perform two forms of slicing: static and dynamic. *Static slices* [84] are generated by leveraging only the static information about the program, i.e., program structure, possible number of threads, possible number of objects, possible sharing of objects, etc. *Dynamic slices* [2] are generated by leveraging both dynamic and static information about the program, i.e., execution histories, execution traces. Based on this distinction, static slices are ideal for program maintenance tasks that involve determining the impact of changes and understanding an existing program while dynamic slices are ideal for debugging tasks that involve determining software fault based on some field data or execution trace information.

Independent of whether or not a slice is calculated using static or dynamic information, we noted in Section 9.1 that slicing can be carried out working backward from the criteria or forward from the criteria. A *backward slice* contains parts of the program that affect the slice criteria, while a *forward slice* contains parts of the program that are affected by the slice criteria. In other words, backward slice is calculated by traversing the control/data flow paths in the program in reverse, while the forward slice is calculated by following the control/data flow paths in the program as they naturally occur.

9.2.1 Program Dependences

Despite the variations, all forms and types of slicing rely on a common set of definitions of program dependences that are based on the static structure of the program. Basically, a program slice can be viewed as the transitive closure of the dependence relation starting from the given slice criteria.

There are two basic forms of program dependences.

- A program point p_t *(dependent)* is *control dependent* on a program point p_e *(dependee)* if p_e can decide if the control can reach p_t during execution.
- A program point p_t *(dependent)* is *data dependent* on a program point p_e *(dependee)* if p_e assigns to a variable v that is used at p_t and there exists a control flow path from p_e to p_t along which v is not assigned at intermediate program points.

In function `bar()` in Figure 19, Line 4 will be executed if `k==0` at Line 3 evaluates to `true`. Similarly, Line 6 will be executed if `k==0` at Line 3 evaluates to `false`. Hence, the Line 3 decides if either Line 4 or Line 6 will be executed, and thus lines 4 and 6 are *control dependent* on Line 3.

In the same function, the value of the expression `v` at Line 7 is determined by the definition of `v` at lines Line 4 and 6. Hence, the expression `v` at Line 7 is *data dependent* on the lines 4 and Line 6.

The initial forms of data and control dependences were introduced by Weiser [84]. Subsequently, Podgurski et.al. [66] extended the definition of control dependence to account for delayed execution due to looping and not mere non-execution. These definitions assumed that the programs have only one end node, i.e., node with zero outgoing edges. Contrary to the latter requirement, most programs (particularly the programs using exception) have multiple or zero (reactive programs) end nodes. In our recent effort [69], we identified these shortcomings and proposed alternative definitions along with algorithms to address these shortcomings.

9.2.2 Aliasing

In the context of languages (such as C, C++, Java) that support pointer/reference variables and heap allocated data, it is common for two variables to point to the same data/memory location, hence, lead to *aliasing*. In such cases, data dependence needs to account for the effect of aliasing. This is illustrated in the function `foo()` in Figure 19. Based merely on the identifiers of the variable `v.o`, we could naively conclude that Line 10 is data dependent on lines 8 and 6. However, as `u` is an alias to `v`, only the definition of `v.o` at Line 9 via *u.o* is used at Line 10. Hence, Line 10 is data dependent only on Line 9 and not on lines 8 and 6. This issue was first identified and addressed by Horwitz et.al. [36].

9.2.3 Slicing Concurrent Java Programs

Most modern applications are concurrent, and it is relatively harder to comprehend concurrent applications. The main issue in comprehension is to statically determine the data flow *(interference dependence)* (due to interleavings) and control flow *(ready dependence)* (due to synchronization) between program points arising due to the scheduling choices during program execution. This issue was initially identified in the context of program dependence/slicing by Krinke [44] and Hatcliff et. al. [32]. These efforts identified the dependences required to enable sound concurrent program slicing. Subsequent efforts [47, 60] proposed approaches based on these new dependences to slice concurrent programs.

In the program in Figure 20, the *savings* `Account` object created at Line 43 is shared between two threads started at lines 46 and 47. In the context of this `Account` object and threads, depending on the runtime schedule,

- the definition of `Account.amount` at Line 8 (Line 12) may affect the use of `Account.amount` at Line 12 (Line 8, Line 5), hence, Line 8 (Line 12) is *interference dependent* on Line 12 (Line 8, Line 5).
- the completion of the monitor acquisition at Line 4 is dependent on the release of the monitor at Line 15 and at Line 10, hence, Line 4 is *ready dependent* on Line 15 and Line 10. Similarly, Line 11 is *ready dependent* on Line 15 and Line 10.
- the completion of the wait on the monitor at Line 6 is controlled by the notification on the same monitor at Line 13, hence, the wait at Line 6 *ready dependent* on the notification at Line 13.

However, the same interference and ready dependences do not apply to the `Account` object created at Line 37 as this object does not escape its thread context, i.e., this object is not shared between threads. Details about this observation and how it can be used to optimize the calculation of interference and ready dependence is presented in our earlier work [70].

9.3 Indus Java Program Slicing Framework

From our experience, we have found that the properties required of a slice vary across different applications of slicing. For example, the program slice required in model checking based program verification applications such as Bandera [19] needs to be executable. On the other hand, executability is not required in applications such as program comprehension via visualization. Even transformations such as slice residualization (i.e., the generation of a new program that contains only the slice) may need to be handled differently for different applications, i.e., destructively updating the original program as opposed to generating a new program. Hence, program slicers need to be modular and flexible (customizable) as opposed to being monolithic and rigid.

Driven by these reasons pertaining to genericity, flexibility, and public consumption, our goal was to implement a general program slicing framework for concurrent Java that could be used to create a customized slicer for diverse applications such as model extraction and program comprehension.

9.3.1 Salient Features

Figure 21 presents the architecture of the Indus slicer. In the sections below, we describe the primary features of the architecture. Due to space constraints of this article, there are many interesting aspects that we do not discuss, but we focus on those features that are most relevant and/or novel.

```
class Account {
  private int amount;

  public synchronized int withdraw(int a) {
    while (amount − a < 0) {
      wait ();
    }
    amount = amount − a;
    return amount;
  }
  public synchronized int deposit(int a) {
    amount = amount + a;
    notifyAll ();
    return amount;
  }
}

class Husband implements Runnable {
  private Account save;

  public Husband(Account account) {
    save = account;
  }
  public void run() {
    save.deposit (90);
  }
}

class Wife implements Runnable {
  private Account save;

  public Wife(Account account) {
    save = account;
  }
  public void run() {
    save.withdraw(10);
    (new Account()). deposit (10);
  }
}

class Home {
  public static void main(String[] s) {
    Account savings = new Account();
    Runnable husband = new Husband(savings);
    Runnable wife = new Wife(savings);
    new Thread(husband). start ();
    new Thread(wife). start ();
  }
}
```

Figure 20: Example to illustrate concurrent Java program slicing. We have omitted the exception around `wait` for brevity.

Intermediate Representation In Indus, Java programs are represented in Jimple [80], a typed three-address representation provided by the SOOT.[6] As as result, every module in Indus operates on the Jimple representation of Java

[6]Soot, a Java OptimizationFramework, is available at http://www.sable.mcgill.ca/soot/ library.

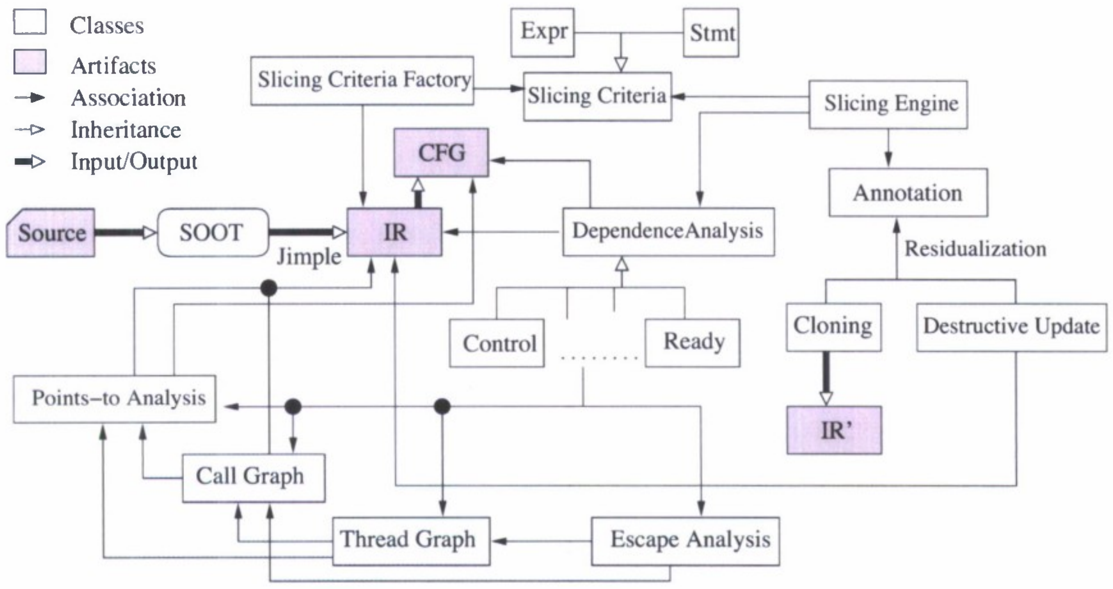


Figure 21: Bird's eye view of classes in Indus Libraries along with the relationships between classes and artifacts.

programs. Hence, any program analyses that is based on Soot can leverage features from Indus with very little effort. Also, provided there is a translator from source to Jimple and/or bytecode to Jimple (as built into SOOT), each module from Indus can be applied to Java source as well as Java bytecodes. Indus leverages the bytecodes to Jimple translation built into SOOT.

Batteries Included In general, program slicing depends, either directly or indirectly, on various forms of dependence analysis that capture the relation between various program points of the program with respect to certain aspects such as data flow, control flow, etc. Thus, when we provide Indus to developers interested in building slicing, analysis, or transformations on top of Indus, we aim to provide a framework where "batteries are included", i.e., all capabilities required to bring the framework to bear on interesting and realistic code bases are already included in the framework. The most common forms of dependences are intra-procedural aliasing-free data dependence, aliasing-based data dependence [68], control dependence [69], (inter-thread data) interference dependence [44], and (inter-thread synchronization) ready dependence [32]. These analyses along with the slicing framework depend on more basic analyses such as escape analysis [70], monitor analysis, and safe-lock analysis [32]. These analyses in turn depend on low-level analyses such as object-flow analysis [67], call graph analysis [9], and thread graph construction [70].

Instead of requiring end-users to develop these analyses from scratch or procuring their implementation from other libraries/projects, Indus provides an implementation (more than one in many cases) of every analysis mentioned above along with other analyses (Figure 21). As Indus libraries are self-contained with respect to analysis, it makes it easier to experiment with program slicing as well as other individual analyses and transformations provided by these libraries.

Loose Coupling, Modularity, and Customizability Every analysis in Indus has two parts — *an interface and an implementation*. If analysis X requires analysis Y then every implementation of analysis X is only coupled with the interface to analysis Y and not to any specific implementation of analysis Y. Hence, each analysis is *modular*, i.e., an implementation of analysis X can be used independently without relying on the implementations of the analysis Y provided by Indus or a third party. As a consequence, it is easy to combine various implementations of analyses to evaluate their combined benefits — a common situation in the application of program analysis. In Indus, this feature is extensively used within the slicing framework and by other analyses to vary the level of accuracy of the result. For example, the slicing algorithm embedded in the `SlicingEngine` class requires a collection of implementations of `IDependencyAnalysis` interface. The sort of dependences provided by these implementations is identified by the associated enumeration value of type `DependenceSort`[7].

[7]This enumeration is available in the upcoming version, and it is currently represented as class constants.

Based on loose coupling and modularity, it is easy to assemble a program analysis pipeline or generate a customized slicer using the components provided by Indus. This is illustrated by the sample command-line applications provided in the Indus distribution, the customized version of the Indus stock slicer used in Bandera, and the use of the analyses provided by Indus in Kaveri.

9.3.2 Advanced Features

Non-SDG based Dependence/Slicing Based on Ottenstein and Ottenstein's result [63], most program dependence and program slicing related efforts are based on *program/system dependence graphs (PDG/SDG)* — program points are represented as nodes and the dependences between program points are represented as directed edges between nodes. Although a PDG accurately captures the dependence relations in an intra-procedural setting, they are altered in an inter-procedural setting [37] to capture dependences arising due to data flow across procedural boundaries, e.g., between formal parameters and arguments and various instances of global variables. Recent efforts pertaining to unconditional jumps [49] and interference dependence further extend SDG. These extensions enable the formulation of slicing as a simple graph reachability problem. However, the downside of these extensions is that non-slicing application will need to extract the required dependences from the dependence graph, hence, contributing to their complexity and cost.

In Indus, instead of maintaining SDGs along with the extension, the logic to handle non-dependence aspects (e.g., argument to parameter binding at call sites) is embedded in the slicing algorithm while the dependences are maintained in their native form (e.g., mappings from variable definition line to variable use line). This approach has following advantages:

- the slicing algorithm can be fine tuned independent of the representation of the dependence information,
- the dependence information can be easily accessed by non-slicing applications without any increase in complexity or cost,
- the cost of constructing and maintaining the dependence graph is eliminated, and
- the maintenance of the slicing algorithm and the dependence analyses is easier due to decreased coupling and increased cohesion.

Program Slicing is Program Analysis Pragmatically, program slicing is often viewed as a transformation that deletes parts of the program P to yield a slice S. However, we have observed that the end goal for which slicing is being applied has a strong influence on whether or not one holds to this view. For example, when program slicing is used for program understanding, testing, and/or debugging, the slice is a mere projection of P that can be captured by annotating P. On the other hand, when program slicing is used to generate executable slices as in Bandera [23], the slice is another program S that is a syntactically correct projection of P.

Based on the above observation, in Indus we consider program slicing as a program analysis — program slicing only calculates the program points that belong to a slice (as in the case of program comprehension). Operations such as residualization of the slice as another program (via destructively updating the program or via cloning the slice) and transforming the slice to be executable are considered as post-slicing transformations. This approach simplifies the slicing algorithm and enables it to be used in various application by merely varying the post-slicing transformations.

Calling Context Sensitive Slicing Consider the example in in Figure 22, suppose the expression `v` in Line 5 is provided as the slice criteria to calculate the backward slice of the program. For sake of simplicity, all of Line 5 will be included in the slice. As the value of `v` depends on the definition in Line 3, all of Line 3 needs to be included in the slice. At this point the slicing algorithm needs to decide which parts of `foo()` needs to be included in the slice. Hence, it descends into `foo()` and decides to include the invocation of `bar()` at Line 10. Similarly, upon descending into `bar()` and including Line 15, the slicing algorithm has two options to exit `bar()`:

- process every calling context leading up to `bar()`, i.e., resume processing at the call sites at Line 4 and at Line 10.
- process only *realizable calling contexts*, i.e., resume processing at the call site at Line 10.

```
1  class CCSS {
2    public static void main(String[] s) {
3      int v = foo();
4      int u = bar();
5      System.out. println ( Integer . toString (v ));
6      int k = foo();
7    }
8
9    static int foo() {
10     return bar ();
11   }
12
13   static int bar() {
14     Long l = new Long(System.currentTimeMillis ());
15     return l. intValue ();
16   }
17 }
```

Figure 22: Example to illustrate calling context sensitive.

The former option is cheap but inaccurate as it calculates the slice based on a semantically unrealizable control flow path (`main()->foo()->bar()->main()`). The latter option requires the slicing algorithm to record the call chain to track realizable paths in order to calculate an accurate slice.

Slicing algorithms that support the first option are said to be *calling context insensitive*. The algorithms that support the second option by keeping track of calling contexts while descending into call sites and tracing back the recorded calling contexts are said to be *calling context sensitive*. Horwitz et.al. [37] proposed the first SDG based calling context sensitive algorithm for sequential programs.

Indus supports calling context insensitive and calling context sensitive slicing of sequential programs. Further, despite calling context sensitive slicing of concurrent programs being exponential, Indus provides a restrictive form of calling context sensitive slicing of concurrent programs that is both efficient and relatively accurate. For more details, please refer to Ranganath's dissertation [68].

Context-restricted Slicing In the program in Figure 22, suppose an user is interested in determining the parts of the program that are affected by the invocation at Line 14 as a result of its execution due to the invocation at Line 3. In such a case, specifying Line 15 as a criteria would result in an inaccurate slice containing lines 3, 4 and 6. The reason being that the slicing algorithm will consider every calling context leading out from (into) the method containing the slice criteria (Line 14) and include every method occurring in these calling contexts in the slice.

We addressed this shortcoming in Indus by enriching the slice criteria with a calling context that can be specified by the user. For example, in the scenario described above, the sequence `[main():*, foo():3, bar():10]` will be supplied as the calling context with the criteria. This restricts the slicing algorithm ascent into invocation sites (e.g., `bar():10`) to only those mentioned in the sequence from right to left. By adopting this approach, the inter-procedural slicing algorithm can trivially leverage auxiliary contextual information to provide accurate call chain specific slices.

This form of slicing is referred to as *context-restricted slicing* and it was introduced by Krinke [48]. Unlike tailoring the slicing algorithm as in Krinke's approach, our approach leverages calling context sensitive and non-graph based slicing along with support for calling context enriched slice criterion to realize context-restriction.

This form of slicing is useful for debugging an application based on an exception stack trace, i.e., an user would like to calculate the slice that affects only the parts of the program occurring on an exception stack trace. In security-related applications, this feature is useful to accurately identify the parts of the programs that affect the data/control flow path between two modules, hence, easily identify any insecure parts of the program.

Scoped Slicing In some applications/situations, it is known that some parts of the program may not contribute interestingly to the slice, e.g., the classes corresponding to the AST nodes of a compiler infrastructure, or the user may want to perform incremental slices to expedite slicing of large programs. In such cases, analyses and slicing can be made more efficient by not considering such parts of the program. For this purpose, the user can limit the analyses and slicing in Indus. This *scoping* feature of Indus can be used to specify a scope of the analysis, i.e., only parts of the system that are within the scope are analyzed. The scope is usually defined in terms of classes, methods, and fields, e.g., $appl.*|Appl$ will limit the analysis to consider only parts of the program belonging to the class `Appl` or the classes with fully qualified name beginning with `appl.`, i.e., belonging to package `appl`).

The slicing framework in Indus supports scoping, and we refer to this form of slicing as *scoped analysis*.[8] Similarly, we refer to any analysis that is *scope-sensitive* as a *scoped analysis*. Currently[9], every analysis implementation in Indus can be executed in a scope sensitive manner by leveraging the general scoping framework.

Scoping	Class	Method	Fields	Size (KB)	Time (sec)	Mem (MB)
none	1198	6136	1973	972	117/539	64/607
auto-slicing	688	2881	879	485	67/462	31/568
auto-analysis	478	1902	597	334	33/142	21/164
manual	436	1856	590	318	30/129	20/150

Table 2: Data from generating sequential executable slices of JReversePro. The data was collected on a Linux box (2GHz/2GB) running Java 1.5.0 with maximum heap space of 1700MB. In the data of the form X/Y, X represents the data for slicing only and Y represents the (overall) data for slicing and the dependent analyses (not transformations). The classes, methods, fields, and bytecode count is inclusive of code pertaining to the application and the required libraries.

Scoping is particularly useful for removing parts of the runtime library that are used during boot strapping and/or for user interface, hence, contribute unnecessarily to slices pertaining to core functionality of the applications. Such a situation occurs in JReversePro[10], a Java Decompiler/ Disassembler consisting of 90 Java classes that amount to 264KB of bytecodes. JReversePro can be used in two modes: command line mode and GUI mode, and the usage mode does not affect its core functionality. However, this separation of functionalities (interface v/s core) cannot be identified in the web of program dependences as they occur in the program. Hence, to illustrate the benefits of scoping, we performed an experiment of sequential slicing on JReversePro in four different settings: 1) *none*, 2) *manual* by eliminating code that connects the GUI part to the core functionality, 3) *auto-slicing* by leveraging the scoping support in Indus only during slicing, and 4) *auto analysis* by leveraging the scoping support in all analyses. In each of these settings, we selected an arbitrary statement that contributes to the core functionality as the slice criterion.

The data from the experiment (as given in Table 2) indicates that both automatic scoped analysis and automatic scoped slicing provide interesting improvements in terms of execution time and memory footprint. Hence, scoping can be used as to scale slicing and other analysis to be applicable to large real-world software.

Interestingly, automatic scoped slicing performs 30% worse than automatic scoped analysis both in terms of time, memory, and slice size. As scoped slicing does not depend on the information pertaining to program points outside the scope, this data indicates that it would be efficient to perform scoped analysis instead of merely performing scoped slicing.

Further, automatic scoped analysis provides slices that are comparable in terms of time, memory, and slice size to those generated via manual scoping. Hence, in situations requiring scoping, automatic scoping can be used for all analysis without much loss of accuracy.

Scoped slicing is also useful in checking for data confinement in the realm of security. For example, one could define a secure scope and calculate a forward slice w.r.t. this scope; if the slice contains any part of the boundary of the scope then it indicates a breach of security.

Although scoped slicing is usually fast and cheap but it may be unsound. In particular, when two program points within the scope may be related by a chain of dependences that involves program points outside the scope. However, such cases are trivially exposed by the inclusion of program points belonging to scope boundary, hence, the user can amend the scope appropriately and incrementally obtain accurate and sound slices.

[8]This notion of restrictive slicing (not analysis) was introduced as *Barrier Slicing* by Krinke [46].
[9]In the latest development version of the libraries.
[10]http://jrevpro.sourceforge.net

Concurrent Java Program Slicing Indus provides various implementations of all dependences mentioned in Section 9.2 to perform both sequential and concurrent slicing of Java programs.

The implementations of interference and ready dependence analysis in Indus rely on a novel, aggressive, and relatively accurate approach [70] to calculate interference and ready dependence based on object flow analysis, and an equivalence class based escape analysis [52, 73]. The approach leverages the escape analysis to rule out cases where the receiver object involved in the dependence relation is not visible outside the creator thread and the object flow analysis to rule out cases where the receiver variables participating in the dependence can never be aliases. Hence, based on this optimization, a slice of the program (in Figure 20) that contains every monitor related operations will contain every invocation to `Account.withdraw()` and `Account.deposit()` except the invocation to `Account.deposit()` at Line 37 — the receiver object in this invocation is does not escape the scope of the creating thread.

Complete Slicing, Chopping, and Control Slicing Besides the support to generate forward and backward slices, the Indus slicing framework, also supports the generation of *complete slices* — a slice that contains parts of the program that affect and are affected by the slice criteria and *every program point in the slice*. The latter aspect distinguishes a complete slice from the mere union of the backward and forward slices w.r.t. the slice criteria. Hence, a complete slice can be perceived as a *software carving* technique that extracts a coherent and syntactically and semantically complete projection of the software w.r.t. the slice criteria.

A common application of slicing/dependences is to determine the programs points that propagate the effect from one given program point to another program point. This set of program points is referred to as the *chop* of the program w.r.t.to the two given program points. As the slicing the algorithm in Indus merely identifies the slice by annotating the program, a chop can be trivially calculated by union-ing the forward slice w.r.t. the first program point with the backward slice w.r.t. the second program point and identifying the parts of the program that contain both the forward and backward annotations.

In many applications of program verification, properties are expressed as "if/when the control reaches this program point, then property ϕ holds". For such applications, the slice should contain parts of the program that are required to ensure that control flow will reach the program point but not to reproduce the behavior of the program when the program point is executed. Such slices are referred to as *control slices*. The Indus program slicing framework supports the generation of control slices by merely enriching the slice criteria with the information that indicates if the slice should preserve the behavior of the program before or after the execution of the slice criteria.

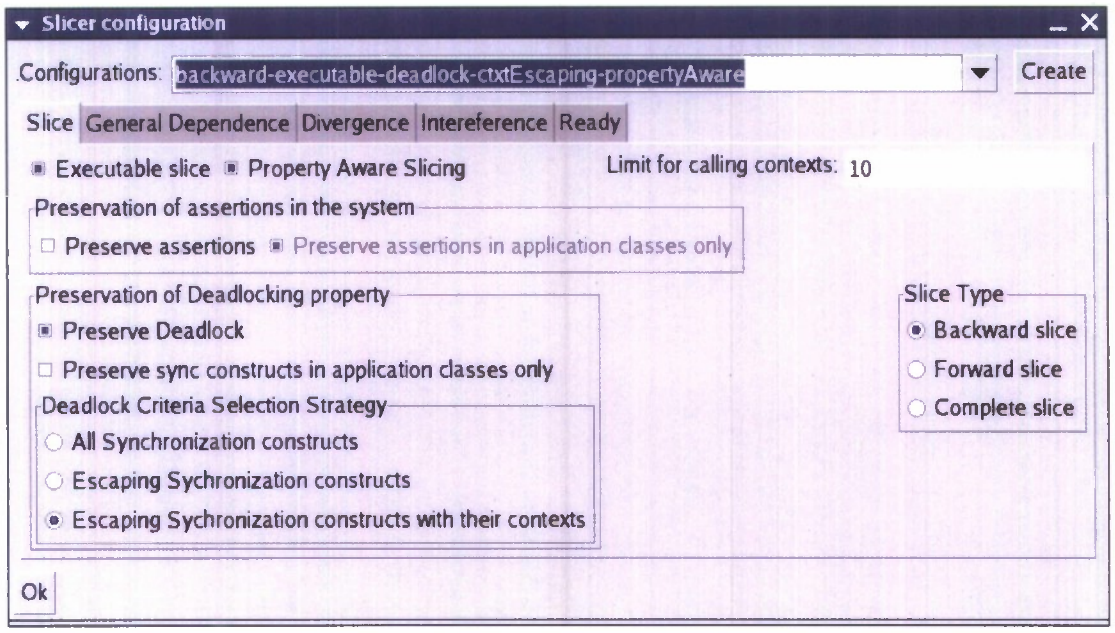
9.3.3 Applying the Framework in Program Verification

As mentioned earlier, program slicing is used in Bandera as a model reduction technique to optimize program verification via model checking. The approach is to identify the program points at which the property will be checked or the program points that are required to preserve a property. These program points are provided as the slice criteria to the slicer to generate an executable backward slice. The slice is residualized by deleting parts of the original program that are not in the slice and by adding necessary parts required to ensure executability. The model generated from the residual sliced program is verified to ensure the program satisfies the property. The details of this approach is documented in earlier efforts [34, 69].

Due to our interests in using Indus for model checking reductions and the observation that deadlock and assertion violations are commonly checked faults, the Indus slicing tool contains pre-packaged features to generate slice criteria required to preserve the deadlocking behavior of the program and/or the assertions in the program with varying levels of precision. The tool also supports a rich configuration that could be used to vary the dependence analyses that need to be used to construct the slice, to control the accuracy of the used dependence analyses, and to select the type of slice. A pre-packaged graphical user interface (shown in Figure 23 can be used to configure the tool as described above.

This tool has been adapted to fit into the tool framework used in Bandera. Similarly, other analyses from Indus have been exposed as tools in the Bandera tool framework. We have successfully applied these tools to various programs consisting of ~144 KB (~10 KLOC) of application bytecodes (~1197 KB of application + library bytecodes).

Recently, we conducted experiments to empirically evaluate the benefits of slicing as a model reduction technique in the context of program verification [24]. Most of the programs (*alarm clock, bounded buffer, disk scheduler, pipeline, sleeping barbers*) considered in the experiments were common examples used to illustrate concurrency in academia. Of the uncommon programs, *RAX* was the distillation of a bug in the NASA remote experiment platform [14], *Mol-*


Slicer configuration

Configurations: backward-executable-deadlock-ctxtEscaping-propertyAware

Create

Slice | General Dependence | Divergence | Intereference | Ready

[✓ Executable slice] [✓ Property Aware Slicing]   Limit for calling contexts: 10

Preservation of assertions in the system
- ☐ Preserve assertions [✓ Preserve assertions in application classes only]

Preservation of Deadlocking property
- [✓ Preserve Deadlock]
- ☐ Preserve sync constructs in application classes only

Deadlock Criteria Selection Strategy
- ○ All Synchronization constructs
- ○ Escaping Sychronization constructs
- [✓ Escaping Sychronization constructs with their contexts]

Slice Type
- [✓ Backward slice]
- ○ Forward slice
- ○ Complete slice

Ok

Figure 23: The pre-packaged graphical user interface to configure the Indus slicer tool.

dyn and *Ray Tracer* were examples from JavaGrande benchmarks[11], and *Siena-Server* and *Siena-Client-Server* were programs built on top of *Siena*, an Internet-scale publish-subscribe infrastructure.[12]

For this publication, we generated calling-context sensitive concurrent backward slices of the above mentioned programs. The deadlocking behavior of the programs were preserved by selecting the synchronization commands in the programs as the slicing criteria. The data from these experiments are summarized in Table 3.

Application	**Classes**	**Fields**	**Methods**	**Statements**	**Time** (seconds)	**Memory** (MB)	**App Size** (bytecodes)
Alarm clock	349/24	1292/11	3434/42	40426/252	7	8	5922
Bounded Buffer	351/25	1297/14	3432/39	40348/191	6	1	5914
Disk Scheduler	348/23	1298/12	3428/38	40500/282	7	8	5697
Pipeline	347/25	1283/5	3421/29	40225/94	0	44	2586
Readers & Writers	348/22	1290/12	3438/43	40476/267	15	39	5919
Sleeping Barbers	347/21	1290/11	3421/29	40322/184	9	41	3139
RAX	347/21	1286/8	3421/29	40229/98	5	44	2501
Moldyn	355/51	1388/99	3478/139	42266/3169	40	52	30598
Ray Tracer	363/70	1359/100	3529/194	42128/3035	34	39	44418
Siena-Server	489/160	1689/259	4929/783	57701/10664	139	111	215366
Siena-Client-Server	489/197	1691/310	4929/927	57722/13677	149	143	210266

Table 3: Summary of data pertaining to slicing concurrent programs in the context of program verification. The data was collected on a 1.4GHz Linux Box running Java 1.5.0_06 with maximum heap space of 512MB. The data in the form X/Y represents the count of the corresponding entity (classes, methods, fields, (Jimple) statements, and bytecodes) before slicing (X) and after slicing (Y). The memory data does include the storage (which was at most 30MB) required by Jimple. The data is inclusive of code pertaining to the application and required libraries.

The data in Table 3 suggests that the Indus slicing tool (framework) yields a reduction of at least 60%, 82%, 82%,

[11] Java Grande Benchmarking Project is available at http://www.epcc.ed.ac.uk/javagrande/.

[12] For more details about these programs, please refer to [24].

and 77% in terms of number of classes, fields, methods, and statements, respectively, in the program (application + libraries) used in our recent experiments; hence, it indicates a high level of accuracy of the embedded slicing algorithm. Similarly, the data also suggests the slicing tool (framework) is efficient both in terms of time and memory as, in the experiments, it could analyze an infrastructure-based application such as Siena-Client-Server within three minutes while requiring less than 256MB of memory on a desktop Linux box.

9.4 Kaveri: A Program Slicing plugin for Eclipse

To the best of our knowledge, to this date there has been only one feature rich slicing tool with a sophisticated UI — the commercial tool named CodeSurfer[13] that is targeted towards C programs. Hence, as part of developing the program slicing framework, we also decided to develop an user interface to an instance of our framework. Given that Eclipse[14] was a open platform that is well accepted by the Java community as a development platform, we choose to develop the slicing UI as a plugin to Eclipse to leverage the extensive Java development support available in Eclipse and to appeal to the Java community.

Kaveri is a plugin that contributes program slicing via an intuitive user interface as a feature to Eclipse platform. The plugin was implemented as a graduate project by Ganeshan Jayaraman from Kansas State University under our supervision.

9.4.1 Architectural Overview

Every aspect of presenting the information via the UI is handled by Kaveri. Trying to be a true source level analysis plugin, Kaveri handles the mapping of between Java and Jimple representations of the programs. This is done by compiling the Java source via the Eclipse Java compiler, generating the Jimple representation from the generated bytecodes, and then constructing a mapping between the Java source and the Jimple representation. This Jimple representation is fed to the Indus program slicing tool (described in Section 9.3.3) to perform slicing.

During slicing, Kaveri leverages the "program slicing is an analysis" feature of Indus to represent the generated slice by annotating/tagging parts of the Jimple representation that belongs to the slice. By combining these annotations with the mapping between Java and Jimple, Kaveri displays the slice in the Eclipse Java Editor view (as shown in the top part of Figure 24). Beyond this, it provides five views that present various dependences existing in the program, hence, aiding the understanding of the generated slice or the program.

9.4.2 Features

The various features and views[15] provided by Kaveri are illustratively described in the following subsections along with a brief outline of how they can be leveraged within Eclipse.

Slice Java Programs As advertised, Kaveri enables the user to perform slicing and view the results at the Java source level. The user can select an arbitrary line in the Java Editor and click on either the backward slice or the forward slice button on the toolbar to perform slicing. The result of slicing is displayed by highlighting the lines of source file included in the slice in green within the Eclipse Java Editor (as illustrated in the top part of Figure 24). Also, the name of any file that contains some part of the slice is decorated in the package explorer[16].

Usually, there will be a few Java statements such that not all parts of these statements are in the slice, e.g., some subexpressions of a statement are omitted from the slice. Such instances are distinguished by highlighting the line in yellow (indicating that only parts of the statement are in the slice) as opposed to green (indicating that all parts of the statement are in the slice). Further, more information about such instances can be retrieved via the Jimple View described next.

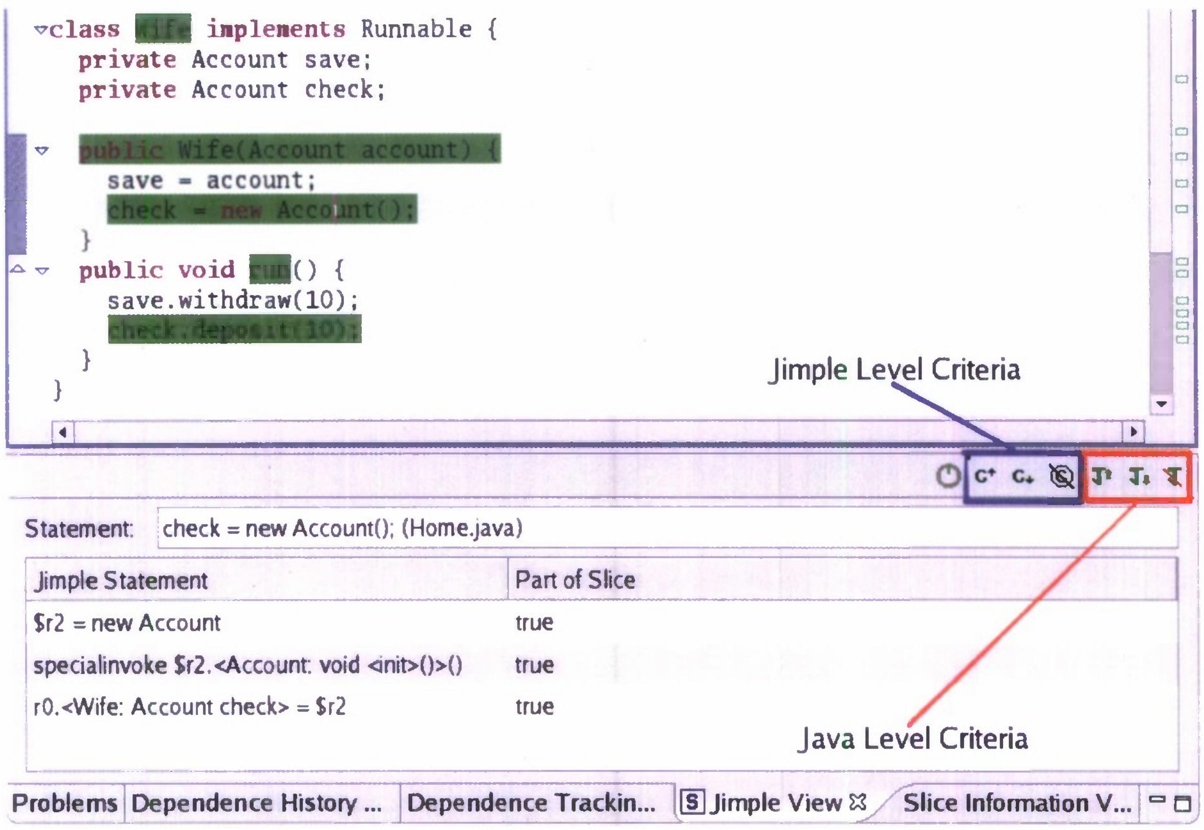
Java to Jimple Mapping The mapping between Java to Jimple generated by Kaveri can be viewed in the *Jimple View*. As shown in Figure 24, this view displays the Jimple statements that collectively represent the Java statement that occurs on the current line in the Java Editor. This view primarily serves four purposes.

[13]CodeSurfer, an analysis and inspection tool for C, is available at http://www.grammatech.com/products/codesurfer/.

[14]Eclipse, an open extensible IDE and tool platform written in Java, is available at http://www.eclipse.org.

[15]The term used in Eclipse to refer to a non-context sensitive display.

[16]A Java specific view in Eclipse that presents project artifacts in hierarchical terms of packages, source files, classes, fields, and methods.



```
class Wife implements Runnable {
  private Account save;
  private Account check;

  public Wife(Account account) {
    save = account;
    check = new Account();
  }
  public void run() {
    save.withdraw(10);
    check.deposit(10);
  }
}
```

Statement: check = new Account(); (Home.java)

Jimple Statement	Part of Slice
$r2 = new Account	true
specialinvoke $r2.<Account: void <init>()>()	true
r0.<Wife: Account check> = $r2	true

Figure 24: A program slice is displayed in the Eclipse Java Editor and the mapping between Java and Jimple representation is displayed the Jimple View.

1. It provides a simple approach to understand how a Java statement is represented as a set of Jimple statements, hence, serving as a Jimple learning aid.

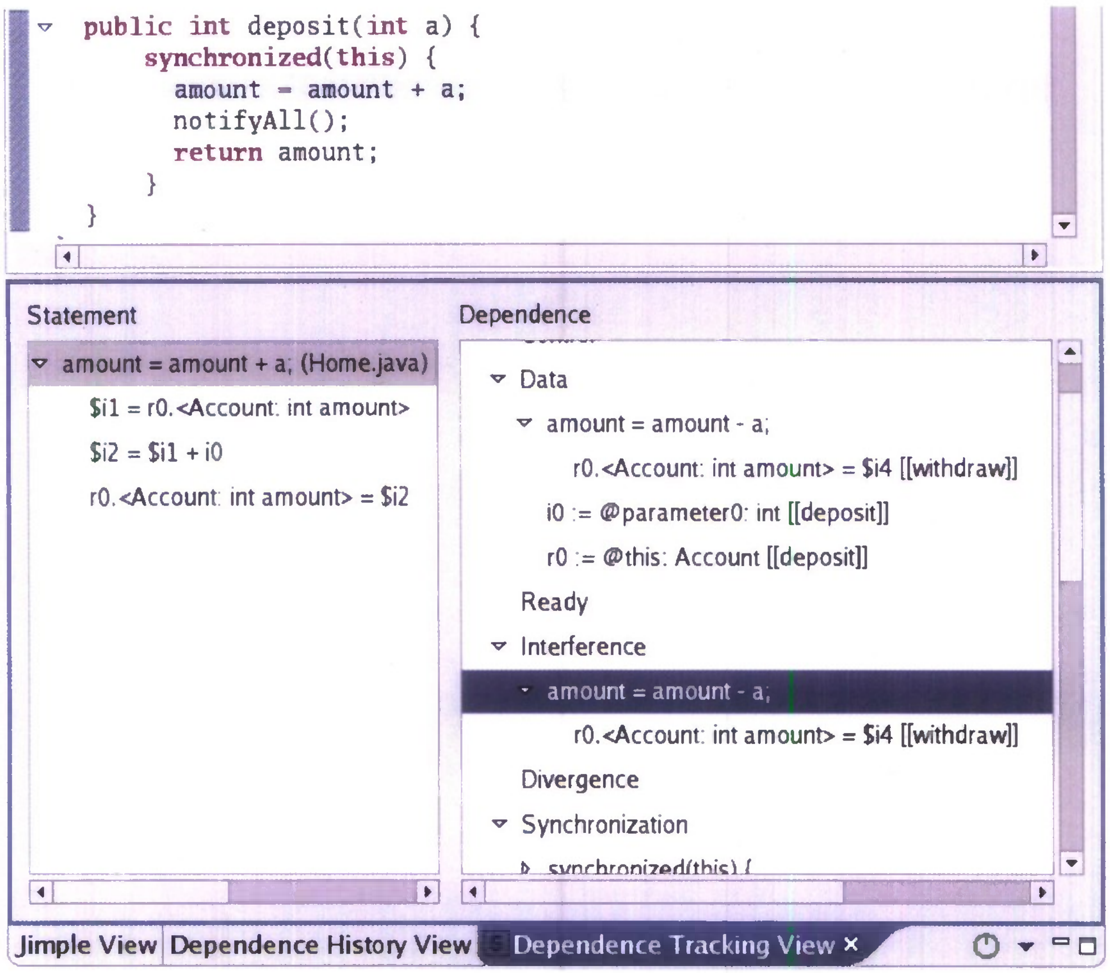
2. It displays the slice at the level of Jimple statements; this information is indicated by displaying *true* or *false* in the "Part of Slice" column against each Jimple statement occurring in the "Jimple Statement" column. This feature is useful in situations when not all parts of a Java statement are included in the slice. In such cases, by clicking on the line highlighted in yellow in the Java editor, the user can view the parts of the statement that are (not) part of the slice.

3. It enables the user to select/unselect slice criteria at the granularity of Java statements. The user can position the cursor at a particular line in Java editor and use the buttons marked as "Java Level Criteria" to select and unselect slice criteria in terms of Java statements (lines). Further, it also provides the fine grained control to select slice criteria to generated control slices (described in Section 9.3.2).

4. An advanced user can select and unselect slice criteria at the granularity of Jimple statements (bytecodes) by selecting the appropriate Jimple statement in view and clicking the buttons marked as "Jimple Level Criteria". This feature is suitable for users who are interested in slicing Java bytecodes.

Kaveri generates the mapping on demand when either slicing is performed or when the user turns on the Jimple View (by clicking on the "power-on" button in the toolbar). As the mapping generation can be a costly operation, the feature to control mapping generation enables the user to use Kaveri in a performance non-intrusive manner.

Dependence Tracking Beyond slicing, users may use Kaveri to view the dependences between various program points in a Java program. This is possible via the *Dependence Tracking View*. The program dependences for the



```
public int deposit(int a) {
    synchronized(this) {
      amount = amount + a;
      notifyAll();
      return amount;
    }
}
```

```
Statement                                Dependence
▽ amount = amount + a; (Home.java)         ▽ Data
    $i1 = r0.<Account: int amount>            ▽ amount = amount - a;
    $i2 = $i1 + i0                                r0.<Account: int amount> = $i4 [[withdraw]]
    r0.<Account: int amount> = $i2              i0 := @parameter0: int [[deposit]]
                                                r0 := @this: Account [[deposit]]
                                             Ready
                                           ▽ Interference
                                              ▽ amount = amount - a;
                                                  r0.<Account: int amount> = $i4 [[withdraw]]
                                             Divergence
                                           ▽ Synchronization
                                              ▷ synchronized(this) {
```

Figure 25: The dependence information for the highlighted statement in the editor is displayed in the Dependence Tracking View by Kaveri.

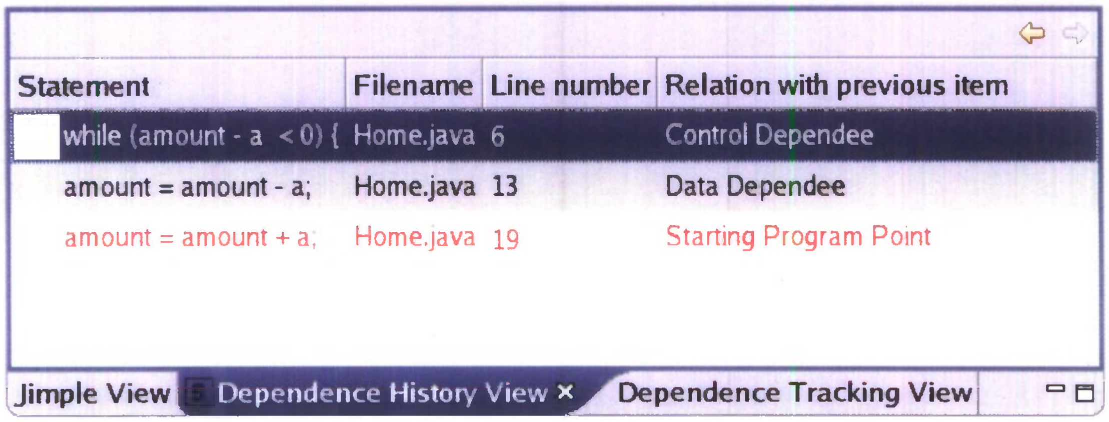
Java statement occurring on the current line in the Java Editor is displayed in this view as show in Figure 25. The dependence information is hierarchically displayed at the granularity of both Java and Jimple statements and in both directions (forward and backward). The user can traverse the dependences by clicking on the dependents/dependees displayed in the "Dependence" column of this view.

For example, in Figure 25, the dependence tracking view indicates that the statement `amount = amount + a;` in `Account.deposit()` is data dependent on the statement `amount = amount - a;` in `Account.withdraw()` as well on the assignments to `this` and `a` in `Account.deposit()`. The latter is a mere identifier based intra-procedural data dependence whereas the former is a reference-based inter-procedural data dependence. Similarly, the view indicates that the same statement is interference dependent on the assignment `amount = amount - a;` in `Account.withdraw()`. Although the interference dependence is valid, the reference-based data dependence is invalid; this is due to the current accuracy limitation of data dependence analysis.

During debugging, it is a common task to traverse through the source code based on the structure of the code. It is also common to backtrack during such traversal and continue forward towards new (or old) program points. For this purpose, tools such as *ctags*[17] and *etags*[18] generate tagging information to expedite such traversal and editors such

[17]Exuberant Ctags: A multilanguage implementation of Ctags, is available at http://ctags.sourceforge.net/.

[18]This is a tagging software available with Emacs.



Statement	Filename	Line number	Relation with previous item
while (amount - a < 0) {	Home.java	6	Control Dependee
amount = amount - a;	Home.java	13	Data Dependee
amount = amount + a;	Home.java	19	Starting Program Point

Figure 26: The history of the dependences "chased" by the user is displayed in the Dependence History View by Kaveri.

as Vim[19] and Emacs[20] support such traversals by leveraging tagging information. Even Eclipse has built-in non-Java specific support for annotation traversal.

Using the dependence tracking view, the user can perform a traversal based on a chain of program dependences. Inspired by the above mentioned support for various forms of code traversal, we provided a similar support to traverse dependences. As part of this support, Kaveri maintains a stack of dependences traversed by the user and this stack can be accessed via the *Dependence History View* as shown in Figure 26. At any point during the traversal, the user can jump to a point in the chain and continue traversing.

In Figure 26, the user has started at line 19 in Home.java and traversed a data dependence to line 13 followed by a control dependence to line 6 in the same file.

Beyond Basics Apart from the basic (select-the-line-and-click-button) mode of slicing, Kaveri provides the following features. These features can be leveraged via the dialog (shown in Figure 27) dedicated to perform slicing in the advanced mode.

- *Perform fine grained slicing*: The user can select Jimple level slice criteria via the Jimple View and then use advanced slicing dialog to perform Jimple statement (bytecodes) level slicing.

- *Perform scoped slicing*: The user can defines various scopes via regular expressions on the fully qualified names of packages, classes, and methods of the software. The regular expressions can be combined with inheritance relation as well. These scopes are managed in a workspace specific manner by Kaveri. As for their usage, the user may choose to enable any of these scopes while performing advanced slicing via the dialog shown in Figure 27.

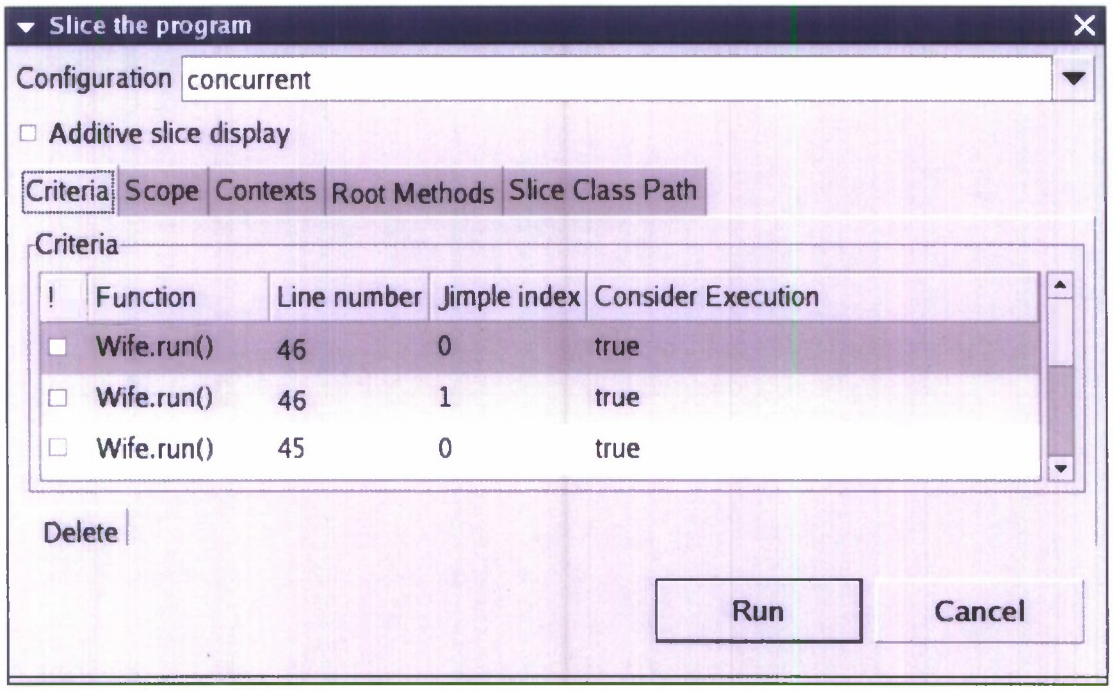
- *Control the classes used for slicing*: In many situations, it may be required to understand a Java program in the context of a particular version of a library. In such situations, the user can switch to advanced mode of slicing and plug in the required library into the "Slice Class Path" before performing slicing.

- *Perform calling context driven slicing*: Kaveri provides support to choose calling contexts via the Eclipse *call hierarchy view*[21]. In the advanced mode of slicing, the user can select the required calling contexts. Before slicing, Kaveri will combine the calling contexts appropriately with the slicing criteria to achieve the effect of calling context driven slicing.

[19] Vim, an extensible editor, is available at http://www.vim.org/.

[20] Emacs, an extensible editor, is available at http://www.gnu.org/software/emacs/.

[21] An view that hierarchically describes the call hierarchy for the selected method.



Configuration: concurrent

!	Function	Line number	Jimple index	Consider Execution
☐	Wife.run()	46	0	true
☐	Wife.run()	46	1	true
☐	Wife.run()	45	0	true

Figure 27: The Kaveri dialog that enables users to select criteria, scoping, and starting calling contexts before slicing.

9.5 Related Work

There have been numerous efforts pertaining to static slicing of sequential programs over the last two decades, and we refer the reader to Frank Tip's survey [79] to learn more about these efforts. In our effort, we have realized various contributions from these efforts along with numerous new advances in Indus, the first publicly available Java program slicing framework.

Our earlier work provides a more formal perspective on several aspects of Indus including novel forms of control dependence using for languages like Java [69], foundations of dependences for concurrent Java [32], and using escape analysis to reduce spurious inter-thread dependences [70].

Recently, Krinke proposed context-restrictive slicing [48] and barrier slicing [46]. We have realized both these forms of slicing in Indus. Further, we have extended the concept of barrier/scope to various program analyses to further improve the scalability of scoped slicing without loss of accuracy.

In the context of static slicing of concurrent programs, almost all previous efforts [47, 59] have been based on PDGs. In contrast, we have proposed the first non-SDG based concurrent slicing algorithm [68].

Bandera [19] was the first effort to apply slicing to Java programs in the context of program verification. Our effort supplements this effort with a robust and accurate Java slicing implementation that can be readily used for the purpose of model reduction in model checking Java programs [24].

To the best of our knowledge, the Java slicer from Univeristy of Passau [31] is the only other program slicer that can handle almost all features of Java. As this slicer is not publicly available, we are unable to compare the catered features of the slicers. Currently, Kaveri is the first and the only publicly available Java program slicing graphical environment.

9.6 Assessment

En route to addressing a very local and specific problem in the realm of program verification, we have developed the Indus Java program slicing framework. This framework is the first and only publicly available general purpose Java slicing implementation. The framework is robust, flexible, and capable of handling almost all features of Java programming language. During this effort, we have found that a non-graph based approach to slicing is feasible and correct, contributed optimizations for interference and ready dependence analysis, proposed alternative definitions for control dependences, and developed new and useful forms of slicing — complete slicing, context-restricted slicing, and scoped slicing. As a result of this effort, we have successfully applied program slicing as a model reduction technique in the context of program verification.

In addition, we have also developed Kaveri – a GUI front-end to Indus with a set of features that are targeted to enable program comprehension via program slicing and program dependences. This tool can serve as a good starting point to introduce program slicing as a program comprehension technique in academia.

Please refer to http://indus.projects.cis.ksu.edu for various publications, software artifacts, and documentation pertaining to Indus and Kaveri.

References

[1] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proceedings of the 1st ACM SIGSOFT symposium FSE*, pages 9–20. ACM Press, 1993.

[2] H. Agarwal and J. H. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, 1990.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[4] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *33rd Principles of Programming Languages (POPL)*, pages 91–102, 2006.

[5] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *11th Static Analysis Symposium (SAS)*, volume 3148 of *LNCS*, pages 100–115. Springer, 2004.

[6] T. Amtoft and A. Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Comp. Prog.*, 64(1):3–28, 2007.

[7] T. Amtoft and A. Banerjee. Verification condition generation for conditional information flow. In *5th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 2–11, 2007. A long version, with proofs, appears as technical report KSU CIS TR 2007-2.

[8] T. Amtoft, J. Hatcliff, E. Rodriguez, Robby, J. Hoag, and D. Greve. Specification and checking of software contracts for conditional information flow (extended version). Technical Report SAnToS-TR2007-5, KSU CIS, 2007. Available at http://www.sireum.org.

[9] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA*, pages 324–341, 1996.

[10] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 2(15):131–177, Mar. 2005.

[11] J. Barnes. *High Integrity Software – the SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[12] G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In R. Foccardi, editor, *CSFW'04*, pages 100–114. IEEE Press, 2004.

[13] J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM TOPLAS*, 7(1):37–61, Jan. 1985.

[14] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, Jul 2000.

[15] CADENA web site. http://cadena.projects.cis.ksu.edu/.

[16] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. In *SIGAda'04, Atlanta, Georgia*, pages 39–46. ACM, Nov. 2004.

[17] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff. Calm and cadena: Metamodeling for component-based product-line development. *Computer*, 39(2):42–50, February 2006.

[18] P. Clements and L. Northrop. *Software Product Lines*. Addison Wesley, 2002.

[19] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, jun 2000.

[20] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *2nd International Conference on Security in Pervasive Computing (SPC 2005)*, volume 3450 of *LNCS*, pages 193–209. Springer, 2005.

[21] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 266–276. ACM Press, 2002.

[22] E. Di Nitto and D. S. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 21st international conference on Software engineering*, pages 13–22. IEEE Computer Society Press, 1999.

[23] M. B. Dwyer and J. Hatcliff. Slicing Software for Model Construction. In *Proceedings of Partial Evaluation and Semantic-Based Program Manipulation (PEPM'99)*, pages 105–118, 1999.

[24] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, Robby, and T. Wallentine. Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2006)*, 2006.

[25] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*. IBM Press, 1997.

[26] D. Gay, P. Levis, D. Culler, and E. Brewer. *nesC 1.1 Language Reference Manual*. http://nescc.sourceforge.net/, May 2003.

[27] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11. ACM Press, 2003.

[28] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[29] A. Gokhale, K. Balasubramanian, and T. Lu. CoSMIC: addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 218–219. ACM Press, 2004.

[30] D. Greve, M. Wilding, and W. M. Vanfleet. A separation kernel formal security policy. In *4th International Workshop on the ACL2 Prover and its Applications (ACL2-2003)*, 2003.

[31] C. Hammer and G. Snelting. An Improved Slicer for Java. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, pages 17–22, 2004.

[32] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Proceedings on the 1999 International Symposium on Static Analysis (SAS'99)*, Lecture Notes in Computer Science, pages 1–18, 1999.

[33] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, volume 841, pages 160–173. IEEE Computer Society Press, May 2003.

[34] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing Software for Model Construction. *Journal of Higher-order and Symbolic Computation*, 13(4):315–353, 2000. A special issue containing selected papers from the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.

[35] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *13th ACM Conference on Computer and Communications Security (CCS'06)*, pages 346–355, 2006.

[36] S. Horwitz, P. Pfeiffer, and T. W. Reps. Dependence Analysis for Pointer Variables. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI'89)*, pages 28–40, 1989.

[37] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Language and Systems*, 12(1):26–60, 1990.

[38] International Organization for Standardization. Information technology – open systems interconnection – basic reference model: The basic model. ISO/IEC 7498-1, 1994.

[39] D. Jackson, M. Thomas, and L. I. Millett, editors. *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, May 2007. Committee on Certifiably Dependable Software Systems, National Research Council.

[40] G. Jung. The type system of calm. Technical Report SAnToS-TR2006-3, Kansas State University, 2006.

[41] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits. Composition and cloning in modeling and metamodeling. *IEEE Transactions on Control Systems Technology*, 12(2):263–278, March 2004.

[42] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[43] A. Kompanek. Modeling a system with acme. http://www.cs.cmu.edu/~acme.

[44] J. Krinke. Static Slicing of Threaded Programs. In *Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, 1998.

[45] J. Krinke. *Advanced Slicing of Sequntial and Concurrent Programs.* PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2003.

[46] J. Krinke. Barrier Slicing and Chopping. In *Proceedings of Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 81–87, 2003.

[47] J. Krinke. Context-Sensitive Slicing of Concurrent Programs. In *Proceedings of ESEC/SIGSOFT FSE'03*, pages 178–187, 2003.

[48] J. Krinke. Context-Sensitivity Matters, But Context Does Not. In *Proceedings of Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 29–35, 2004.

[49] S. Kumar and S. Horwitz. Better Slicing of Programs with Jumps and Switches. In *Proceedings of Fundamental Approaches of Software Engineering (FASE'02)*, pages 96–112, 2002.

[50] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Y. Thomason, G. G. Nordstrom, and P. Volgyesi. The generic modeling environment. In *Proceedings of the Workshop on Intelligent Signal Processing*, May 2001.

[51] Á. Lédeczi, G. Nordstrom, G. Karsai, P. Völgyesi, and M. Maróti. On metamodel composition. In *Proceedings of the 2001 IEEE International Conference on Control Applications (CCA)*, pages 756–760, 2001.

[52] D. Liang and M. J. Harrold. Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses. In *Proceedings 8th International Static Analysis Symposium (SAS 2001)*, volume 2126, pages 279–298, July 2001.

[53] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 3–14. ACM Press, 1996.

[54] V. Matena, S. Krishnan, L. DeMichiel, and B. Stearns. *Applying Enterprise JavaBeans*. Addison Wesley, 2003.

[55] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 24–32. ACM Press, 1996.

[56] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[57] R. Monroe. Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, CMU School of Computer Science, September 2000.

[58] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL'99, San Antonio, Texas*, pages 228–241. ACM Press, 1999.

[59] M. G. Nanda. *Slicing Concurrent Java Programs: Issues and Solutions*. PhD thesis, Indian Institute of Technology, Bombay, November 2001.

[60] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'00)*, pages 180–190, 2000.

[61] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *11th European Symposium on Research in Computer Security (ESORICS'06)*, volume 4189 of *LNCS*, pages 279–296. Springer, 2006.

[62] Object Management Group. *OMG formal/06-04-01 (CORBA Component Model Specification, v4.0)*, April 2006.

[63] K. Ottenstein and L. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments (PPDE'84)*, pages 177–184, 1984.

[64] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (Lecture Notes in Computer Science 607)*, 1992.

[65] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[66] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(8):965–979, 1990.

[67] V. P. Ranganath. Object-Flow Analysis for Optimizing Finite-State Models of Java Software. Master's thesis, Department of Computing and Information Science, Kansas State University, 2002.

[68] V. P. Ranganath. *Scalable and Accurate Approaches to Program Dependence Analysis, Slicing, and Verification of Concurrent Object Oriented Programs*. PhD thesis, Department of Computing and Information Science, Kansas State University, 2006. Work In Progress.

[69] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A New Foundation For Control-Dependence and Slicing for Modern Program Structures. In *Programming Languages and Systems, Proceedings of 14th European Symposium on Programming, ESOP 2005*, 2005. Extended version is available at http://projects.cis.ksu.edu/docman/?group_id=12.

[70] V. P. Ranganath and J. Hatcliff. Pruning Interference and Ready Dependences for Slicing Concurrent Java Programs. In *Proceedings of Compiler Construction (CC'04)*, volume 2985 of *Lecture Notes in Computer Science*, pages 39–56, 2004.

[71] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 432–448, 2004.

[72] B. Rossebo, P. Oman, J. Alves-Foss, R. Blue, and P. Jaszkowiak. Using SPARK-Ada to model and verify a MILS message router. In *Proceedings of the International Symposium on Secure Software Engineering*, 2006.

[73] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, jun 2000.

[74] J. Rushby. The design and verification of secure systems. In *8th ACM Symposium on Operating Systems Principles*, volume 15(5), pages 12–21, 1981.

[75] M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, 1996.

[76] V. Simonet. Flow Caml in a nutshell. In G. Hutton, editor, *First APPSEM-II workshop*, pages 152–165, Mar. 2003.

[77] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM TOSEM*, 15(4):410–457, Oct. 2006.

[78] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *12th Static Analysis Symposium*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.

[79] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995. Surveys the state-of-the-art in program slicing and gives many references to the literature.

[80] R. Vallée-Rai. SOOT: A Java Bytecode Optimization Framework. Master's thesis, School of Computer Science, McGill University, 2000.

[81] M. Vanfleet, J. Luke, R. W. Beckwith, C. Taylor, B. Calloni, and G. Uchenick. MILS: Architecture for high-assurance embedded computing. *CrossTalk: The Journal of Defense Software Engineering*, August 2005.

[82] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–188, 1996.

[83] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison Wesley, 2nd edition, August 2003.

[84] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[85] xADL 2.0 schemas. http://www.isr.uci.edu/projects/xarchuci/ext-overview.html.

[86] Sireum website. http://www.sireum.org.